AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE
Wydział Fizyki i Informatyki Stosowanej

Praca magisterska

Łukasz Pawlik

kierunek studiów: **Informatyka Stosowana**

Opracowanie systemu szybkiego przesyłania danych z wykorzystaniem standardu USB

Promotor: **dr inż. Bartosz Mindur**

Kraków, 2015

Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie i nie korzystałem ze źródeł innych niż wymienione w pracy.

.....

(czytelny podpis)

Merytoryczna ocena pracy przez Opiekuna:

Ocena końcowa pracy przez Opiekuna:

Data: Podpis: .....

Merytoryczna ocena pracy przez Recenzenta:

Ocena końcowa pracy przez Recenzenta:

Data: Podpis: .....

Spis treści

Spis elementów implementacji

I. Wstęp

W dobie dzisiejszych możliwości rosną również wymagania i oczekiwania. Dotyczy to prawie każdego aspektu życia, począwszy od spraw osobistych, poprzez finansowe, aż po możliwości uzyskiwania danych. Rozwój technologii, zwłaszcza w zakresie elektroniki, w przeciągu minionego ćwierćwiecza stał się imponujący. Udało się doprowadzić różne skomplikowane układy scalone do rozmiarów wielkości pudełka zapałek, których złożoność obliczeniowa przekracza najśmielsze oczekiwania z przed kilku lat. Dla współczesnego człowieka problemem okazuje się czas, zawsze jest go za mało w praktycznie każdej sferze życia. Jest on również istotny w rozwoju technologii oraz uzyskiwaniu konkretnych danych pomiarowych.

Dzisiejsze urządzenia pomiarowe laboratoryjne jak i medyczne (oraz inne) mają za zadanie zbieranie i przesyłanie danych celem późniejszej wizualizacji lub analizy za pomocą dedykowanego oprogramowania. Dane te z reguły są wysyłane w określonym dla danego urządzenia formacie, a co za tym idzie ich rozmiar musi być większy niż w wypadku danych wysyłanych bajt po bajcie.

Człowiek, odkąd powstał pierwszy komputer, rozpoczął walkę z problemem komunikacji między urządzeniami. Zostało opracowanych wiele standardów komunikacji, niektóre projekty zostały skazane na zapomnienie, inne odniosły sukces i zdominowały urządzenia. Takim standardem jest z pewnością USB, czyli Uniwersalna Magistrala Szeregowa (dokładny opis wraz z rysem historycznym w rozdziale II.2). Na dzień dzisiejszy większość urządzeń dla których konieczna jest komunikacja z komputerem osobistym (wyłączając sieci Ethernetowe [1] oraz WiFi[2]), posiada interfejs USB.

Celem niniejszej pracy było opracowanie szybkiego przesyłania danych używając standardu USB. Przez szybkie rozumiana jest tutaj prędkość nie mniejsza niż 140Mbit/s (17,5 MB/s) [3].

Pierwszym zadaniem, które wykonano przed rozpoczęciem realizacji implementacji było wybranie hardwaru, pełniącego role urządzenia testowego (nie jest możliwe wykonanie testów przesyłu danych po interfejsie USB bez urządzenia odbierającego dane). Dokładny opis wybranego urządzenie został umieszczony w rozdziale V. Kolejnym zadaniem był odpowiedni wybór biblioteki spełniającej założenia projektu. Istnieje wiele implementacji bibliotek udostępniających API [4] dla interfejsu USB. W rozdziale III szczegółowo scharakteryzowano dwie najczęściej używane, z których pierwsza używana jest zarówno pod systemem Windows jak i pod systemem Unix, natomiast użycie drugiej z nich możliwe jest tylko pod

[1] Ethernet - standard sieci lokalnych

[2] WiFi - technologia sieci bezprzewodowych

[3] 1 B = 8 bit

[4] API (ang. Application Programming Interface) - zestaw funkcji oraz innych narzędzi umożliwiających wykorzystanie możliwości danej biblioteki

systemami Windows. Dodatkowo zostało przedstawione krótkie uzasadnienie wyboru.

Kolejnym etapem podczas projektowania oprogramowania było dokładne zaplanowanie jak przebiegać będą testy. Warto zwrócić uwagę, że ich wykonanie bez urządzenia zewnętrznego było niemożliwe. Stało się konieczne opracowanie wejściowego jak i zwracanego formatu danych dla każdego testu. Po opracowaniu podstawowych wymagań i testowych restrykcji dla aplikacji możliwe było przystąpienie do jej pisania. Opis wraz z implementacją znajduje się w rozdziale VI.

Ostatnim i zarazem najważniejszym elementem pracy było zestawienie otrzymanych wyników wraz z oczekiwaniami. Całość została opisana w rozdziale VII i zawiera konkretne wnioski na temat realizacji projektu oraz ogólnych możliwości interfejsu USB. W rozdziale zawarte jest dodatkowo zestawienie i interpretacja otrzymanych wyników w zakresie synchronicznego oraz asynchronicznego przesyłania danych za pomocą USB. Została w nim zaprezentowana dodatkowa symulacja, aby sprawdzić zachowanie interfejsu w mniej dostępnym środowisku.

II. Motywacje oraz podstawowe założenia projektu

Podstawową motywacją rozpoczęcia pracy nad projektem była chęć zdobycia wiedzy w zakresie komunikacji opartej na interfejsach szeregowych. Jest to wiedza, która może zostać wykorzystana w przyszłości, a tego typu projekty z pewnością prowadzą do samodoskonalenia umiejętności człowieka, który się ich podejmie. Ważnym elementem było zaczerpniecie jak największej wiedzy na temat interfejsów szeregowych oraz samego USB.

Głównym celem projektu było osiągnięcie odpowiedniej prędkości przesyłu danych pomiędzy komputerem osobistym a urządzeniem zewnętrznym. Prędkością oczekiwaną było 140 Mbit/s co w przeliczeniu na megabajty daje 17,5 MB/s w obu kierunkach. Projekt miał za zadanie posłużyć jako program umożliwiający przesyłanie zadanej ilości danych z prędkością nie mniejszą od oczekiwanej. Mogłoby to znaleźć zastosowanie w wielu dziedzinach życia, wszędzie tam gdzie urządzenie udostępniałoby interfejs USB i umożliwiałoby przesłanie danych poprzez wykorzystanie jego możliwości. Doskonałym przykładem, w których możliwe byłoby zastosowanie projektu jest oprogramowanie powiązane z odczytem danych z urządzeń medycznych (np. tomografu, rezonansu magnetycznego), które przesyłają dane w formacie DICOM[1].

II.1. Opis interfejsów szeregowych oraz ich zastosowanie

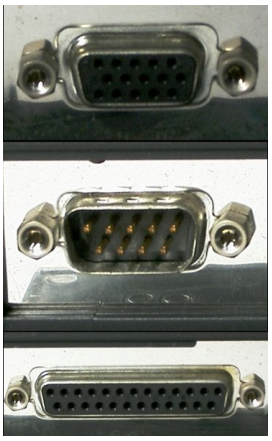
Porty szeregowe w kontekście informatycznym są to fizyczne interfejsy umożliwiające przesłanie danych z lub do urządzenia w jednostce czasu jeden za drugim. Cechy interfejsów szeregowych można opisać za pomocą trzech prostych punktów:

- prostota
- wydajność
- trudność implementacji.

Do najprostszych interfejsów szeregowych zaliczamy z pewnością interfejsy takie jak One-Wire oraz RS232C (kabel przedstawiony na rysunku II.1).

[1] DICOM (ang. Digital Imaging and Communications in Medicine) - standard opracowany celem ujednolicenia przesyłania obrazów w medycynie

Rys. II.1. *RS232C [1]*



Rys. II.2. *Rodzina RS232 [2]*

Złącza typu RS232C można jeszcze spotkać jako interfejs komunikacyjny z drukarkami starego typu, niemniej został on wyparty przez nowsze interfejsy (takie jak np. USB).

Interfejsami najbardziej wydajnymi są USB oraz Ethernet. Są to stosunkowo nowo opracowane (w porównaniu do RS232) i nadal rozwijane interfejsy. Dokładny opis standardu USB znajduje się w rozdziale II.2. Warto również wspomnieć, że najtrudniejszym do zaimplementowania protokołem/interfejsem przesyłania danych jest Bluetooth (w wypadku kategoryzacji według takiego wyznacznika). Służy on do bezprzewodowej komunikacji na bardzo krótkim dystansie.

Podsumowując, interfejsy szeregowe umożliwiają komunikację miedzy urządzeniami, z których jeden najczęściej jest komputerem osobistym. Implementacja każdego interfejsu jest na swój sposób złożona. Kategoryzacja jest możliwa na podstawie różnych wielkości. W tym rozdziale zostały podzielone według złożoności interfejsu (w sensie architektury), wydajności oraz trudności implementacji.

II.2. Opis USB

Uniwersalna Magistrala Szeregowa jest to standard opracowany w latach 90. XX w. definiujący jakie kable, złącza oraz protokoły mają być używane podczas połączenia, komunikacji oraz definiuje sposób zasilania pomiędzy komputerem i urządzeniem elektronicznym. USB zostało zaprojektowane aby ułatwić połączenia standardowych elektronicznych urządzeń takich jak klawiatury, myszki, drukarki, aparaty cyfrowe, dyski przenośne do komputerów osobistych. Wszystkie te urządzenia są dodatkowo zasilane również za pomocą tego portu. Z czasem stało się to wspólne również dla innych urządzeń takich jak smartphony, palmtopy oraz konsole wideo.[3, 4, 5]

USB szybko zastąpiło porty szeregowe oraz równoległe podobnie jak inne urządzenia będące źródłem zasilania dla innych układów scalonych. Istnieją trzy podstawowe rodzaje

złączy USB, dla których kryterium podziału stanowi wielkość (wtyczki). Najstarszy rozmiar (używany np. w pendrive'ach - typ A) występuje w standardach USB1.1, USB2.0, USB3.0, mini-USB (początkowo tylko dla złącza typu B, jak w wypadku wielu aparatów cyfrowych) oraz mikro-USB występuje również w trzech wariantach dla USB1.1, USB2.0, USB3.0 (dla przykładu używany w nowych telefonach komórkowych).

W przeciwieństwie do innych kabli do przesyłu danych (np. Ethernet, HDMI) każdy koniec kabla zakończony jest innym typem złącza (typem A lub typem B). Tylko złącze typu A jest odpowiedzialne za dostarczenie zasilania do złącza typu B. Zostało ono zaprojektowane w taki sposób aby uniknąć elektrycznych przeciążeń, a co za tym idzie uszkodzeniu urządzenia. Istnieją również kable ze złączami typu A na obu końcach, ale nie należą do popularnych (i należy postępować z nimi ostrożnie). Kable USB maja zazwyczaj złącze typu A (męskie) z jednej strony oraz złącze typu B z drugiej (również męskie) oraz wejście w komputerze lub urządzeniu elektronicznym (żeńskie). W przyjętej praktyce złącze typu A jest zazwyczaj największej (z możliwych) wielkości, natomiast B w zależności od potrzeb użycia kabla (full, mini, micro). [3, 4, 5, 6]

Rys. II.3. *Logo standardu USB2.0 oraz USB3.0 [7]*

USB zapoczątkowało w 1994 siedem firm: Compaq, DEC, IBM, Intel, Microsoft, NEC, Nortel. Celem było uproszczenie podłączenia zewnętrznych urządzeń do komputera zastępując stare złącza w płytach głównych wprowadzając rozwiązania na problemy znalezione w starych oraz upraszczając software. Pierwszy układ scalony wspierający USB został wyprodukowany przez Intel 1995r.

Pierwsza oficjalna wersja standardu USB została wydana w styczniu 1996r. USB1.0 charakteryzowała prędkość 1,5 Mbit/s (Low Speed) oraz 12 Mbit/s (Full Speed). Nie pozwalał jednak na używanie przedłużaczy kabli, wynikało to z limitów zasilania. Kilka powstałych wypuszczono na rynek na chwile przed wydaniem standardu USB1.1 w sierpniu 1998r. W USB1.1 poprawiono kilka błędów znalezionych w USB1.0 i był to pierwszy standard, który został oficjalnie zaimplementowany w standardowych komputerach osobistych. [3]

USB2.0 zostało wydane w kwietniu 2000r. udostępniając maksymalny przesył sygnału rzędu 480 Mbit/s (60MB/s) nazwany High Speed (USB1.x za pomocą Full Speed umożliwiał przesył rzędu 12Mbit/s). Biorąc pod uwagę zależności dostępu do magistrali przepustowość High Speed ogranicza się do 280 Mbit/s (35 MB/s). Przyszłe modyfikacje do specyfikacji USB zostały zaimplementowane przez "Engineering Change Notitices" (ECN). Najważniej-

sze z ECNów zostały dołączone do specyfikacji USB2.0, dostępnej na stronie internetowej USB.org. Przykłady ECNow to Złącze Mini-A oraz Mini-B: wydane w październiku 2000r.[4]

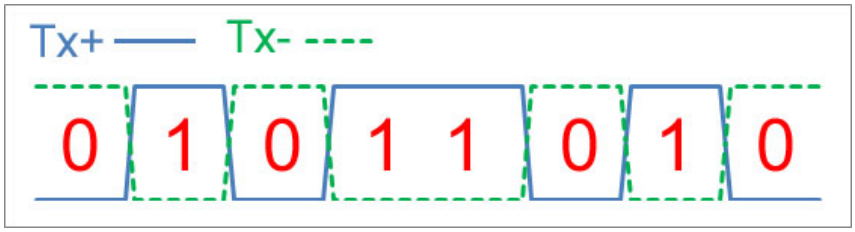
Standard USB3.0 został wydany w listopadzie 2008r. definiując zupełnie nowy tryb "SuperSpeed". Port USB zwyczajowo jest w kolorze niebieskim i jest on kompatybilny z urządzeniami USB2.0 oraz kablami. Dokładnie 17 listopada 2008r. ogłoszono iż specyfikacja dla wersji 3.0 została całkowicie ukończona i została zaakceptowana przez "USB Implementers Forum" (USB-IF), czyli głównej instytucji zajmującej się specyfikacjami standardu USB. To pozwoliło na szybkie udostępnienie standardu deweloperom. Nowa magistrala "SuperSpeed" dostarcza czwarty typ transferu z możliwością przesyłania sygnału z prędkością 5GBit/s, ale poprzez użycie kodowania 8b/10b przepustowość wynosi 4Gbit/s. Specyfikacja uznaje za zasadne osiągniecie prędkości w okolicach 3,2 Gbit/s (400 MB/s), co w założeniach powinno się zwiększać wraz z rozwijaniem hardwaru. Komunikacja dla SuperSpeed odbywa się w obu kierunkach (kierunek nie jest naprzemienny i nie jest kontrolowany przez hosta, jak to ma miejsce do wersji USB2.0). Podobnie jak w poprzednich wersjach standardu, porty USB3.0 działają dwóch wariantach zasilania: niskiego poboru mocy (low-power: 150mA) oraz wysokiego poboru mocy (high-power: 900mA). Zapewniając odpowiedni, jednocześnie pozwalają na przesył danych z prędkością SuperSpeed. Została dodatkowo zdefiniowana specyfikacja zasilania (w wersji 1.2 wydana w w grudniu 2010r.), która zwiększała dopuszczalny pobór mocy do 1,5A, ale nie pozwala na współbieżne przesyłanie danych. Specyfikacja wymaga aby fizyczne porty same w sobie były w stanie obsłużyć 5A, ale ogranicza pobór do 1,5 A. [5]

W styczniu 2013r. w prasie pojawiły się informacje o planach udoskonalenia standardu USB3.0 do 10Gbit/s. Zakończyło się to stworzeniem nowej wersji standardu - USB3.1. Wersja ta została wydana 31 lipca 2013r. wprowadzając szybszy typ przesyłania danych zwany "SuperSpeed USB 10 Gbit/s". Zaprezentowano również nowe logo stylizowane na zasadzie "Superspeed+". Standard USB3.1 zwiększył szybkość przesyłu sygnału do 10Gbit/s. Udało się też zredukować obciążenie łącza do 3% dzięki zmianie kodowania na 128b/132b. Przy pierwszych testach prędkości USB3.1 udało się uzyskać wynik na poziomie 7,2Gbit/s. Standard USB3.1 jest wstecznie kompatybilny ze standardem USB3.0 oraz USB2.0. [5]

Poszczególne rodzaje kodowania dla standardów USB3.0 oraz USB3.1 to kodowanie 8bitów na 10 bitów oraz 128 bitów na 132 bity. Jeśli chodzi o przesyłanie danych to do momentu wprowadzenia Differential signaling (polega na wysłaniu dwóch przeciwnych sobie sygnałów TX+ oraz TX- do zewnętrznego odbiornika, który aby zniwelować zaszumienie odejmuje wartość RX- od obydwu kanałów) przesyłanie danych na długie odległości lub wysyłanie danych z większą prędkością było awykonalne. Przykład wraz z kolejnymi etapami widoczny na rys II.4, II.5, II.6.

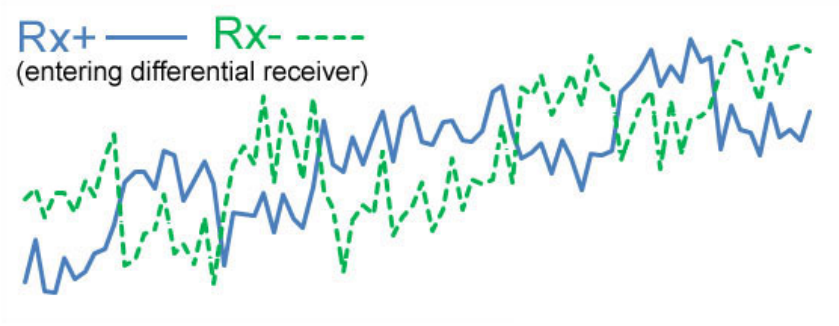
W czasach kiedy elektronika miała swoje początki, zarówno nadawca jak i odbiorca współdzielili to samo uziemienie (były częścią jednego środowiska), więc nie było problemów aby

zmierzyć napięcie sygnału wejściowego. Więc jeżeli założenie było iż 0V reprezentuje wartość bitu równą ’0’ a wartość +5V reprezentuje wartość bitu równą ’1’ oraz zegar był również wspólny dla nadajnika oraz odbiorcy, odbiorca świetnie potrafił odczytać nadawany sygnał (np. rys. II.4).

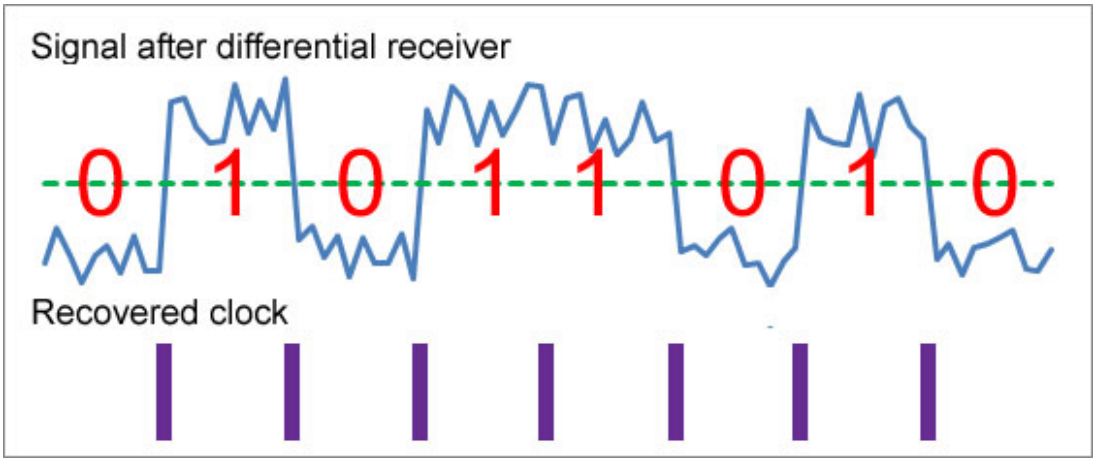


Rys. II.4. *Przykład czystego (bez zakłóceń) sygnału transmitowanego [8]*

Problem pojawia się w momencie kiedy odbiornik znajduje się poza środowiskiem nadawcy (nadawca i odbiorca nie współdzielą uziemienia, zegara, etc.). Rys. II.5 przedstawia sygnał odebrany po stronie odbiornika (jest to ten sam sygnał co na rys. II.4). Przez brak współdzielenia uziemienia oraz zegara przez nadajnik oraz odbiornik sygnał stał się nieczytelny po stronie odbiornika.



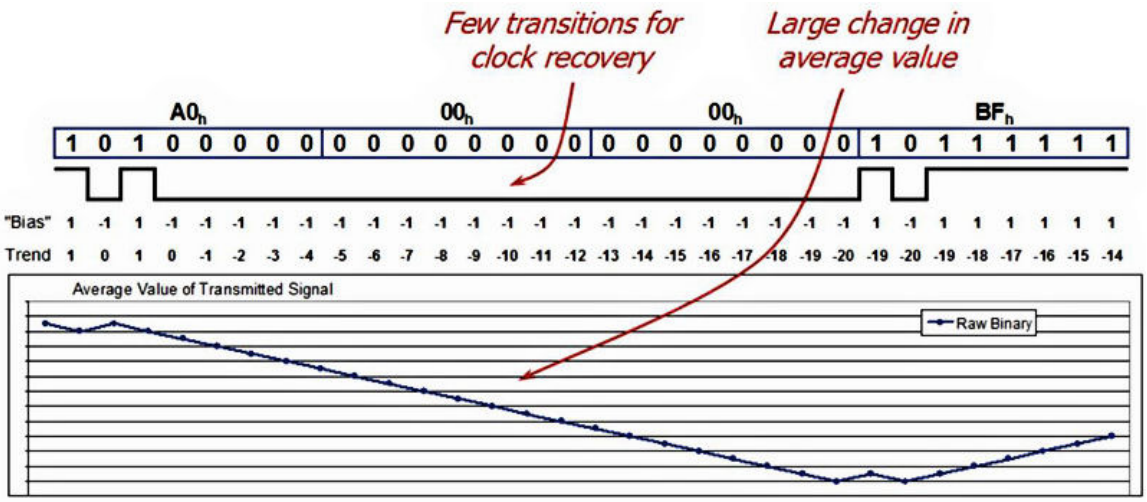
Rys. II.5. *Przykład odebranego sygnału (wraz z zakłóceniami) [9]*



Rys. II.6. *Przykład sygnału po oczyszczeniu z szumów [10]*

Rozwiązaniem problemu przesyłu danych między dwoma zewnętrznymi elektronicznymi urządzeniami okazało się zaprojektowanie differential receiver'a, czyli odbiornika odbierającego dwa sygnały RX+ oraz RX-. Poziom sygnału jest mierzony na podstawie ich złożenia. Skrętka[2] gwarantuje, że przesunięcie napięcia (rys. II.5) dla obu kanałów jest takie samo, a więc różnica pomiędzy jest stała i jest możliwa do odczytu przez odbiornik (rys. II.6). Dodatkowo odbiornik jest w stanie odczytać sygnał zegara na podstawie szybkości zmian sygnału (rys. II.6).

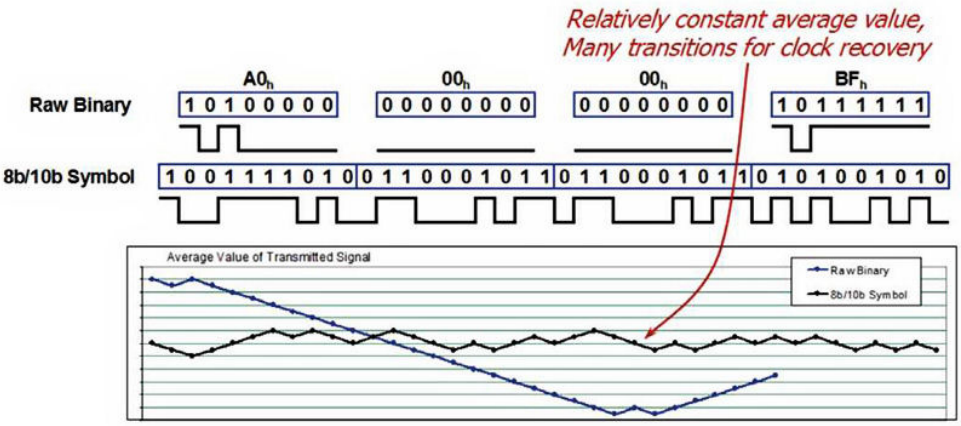
W przypadku wysyłania większej ilości danych pojawia się kolejny problem. Jeżeli ilość tych samych wartości bitu (zer lub jedynek) występujących po sobie jest zbyt duża, odbiornik może nie być w stanie odczytać sygnału zegara lub zrobić to błędnie. Rys. II.7 zawiera przykładowy zestaw wysyłania większej ilości zer, jak widzimy, w drugim i trzecim pakiecie danych wysyłane są same zera, a ilość przeskoków sygnału (zmian 0->1 lub 1->0) jest znikoma i obliczenie sygnału zegara może być problematyczne dla odbiornika.



Rys. II.7. *Przykład sekwencji danych bez kodowania [11]*

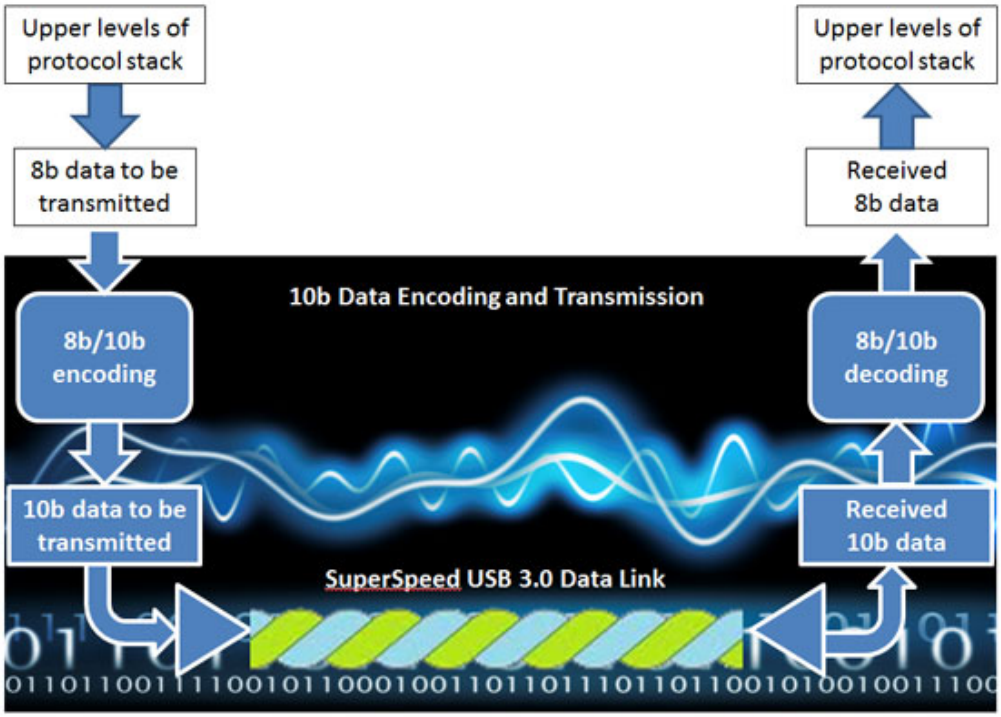
Rozwiązaniem tego problemu jest kodowanie 8b/10b, dzięki któremu ilość zmian sygnału z zera na jeden oscyluje w okolicach 50% w stosunku do zmian z jeden na zero, dzięki czemu obliczenie zegara na odbiorniku nie jest problematyczne. Przykład sygnału po zastosowaniu kodowania 8b/10b jest widoczny na rys. II.8.

[2] Skrętka - w kontekście pary skręconych przewodów



Rys. II.8. *Przykład sekwencji po zakodowaniu [12]*

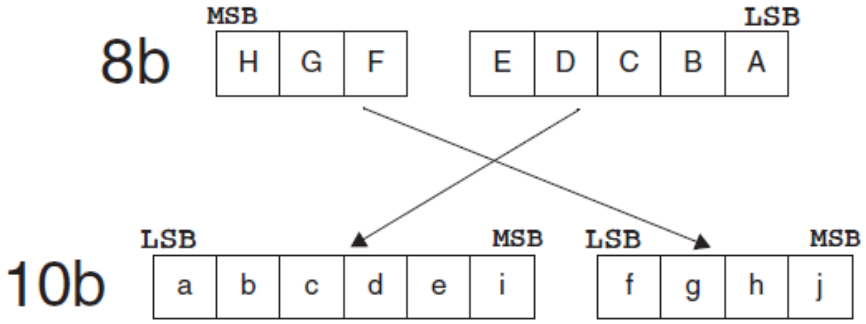
Rys. II.9 przedstawia działanie USB3.0 wraz z kodowaniem oraz komunikacją przez warstwę fizyczną. Najpierw dane do wysłania przekonwertowywane są na z postaci 8-bitowej na postać 10-bitową, następnie wysyłane do odbiornika, gdzie następuje konwersja odwrotna do postaci 8-bitowej.



Rys. II.9. *Przykład wysyłania danych przez warstwę fizyczną wykorzystując kodowanie [13]*

Kodowanie 8b/10b w swojej budowie jest kombinacją dwóch innych: 5b/6b oraz 3b/4b z zachowaniem odpowiedniej zasady ich łączenia. Cały algorytm tworzenia przedstawia się następująco:

1. 8 bitów jako dane wejściowe: HGFEDCBA[3],
2. na ostatnich 5 bitach (EDCBA) wykonujemy kodowanie 5b/6b (dochodzi dodatkowy bit),
3. na pierwszych trzech bitach (HGF) wykonujemy kodowanie 3b/4b (dochodzi kolejny bit),
4. całość łączona do postaci abcdeifghj[4] tworząc rezultat (przedstawiony na rys. II.10).



Rys. II.10. *Przykład mapowania w kodowaniu 8b/10b [14]*

Dwa ostatnie bity ustala się oddzielnie dla tych dwóch kodowań, na podstawie współczynnika RD[5], czyli różnicy pomiędzy ilością jedynek i zer w ciągu. Jak zostało wspomniane wyżej, założenie całego kodowania jest takie aby, ilość zer całości przesyłanych danych oscylowała w okolicach 50%, więc różnica (RD) dla jednego zestawu 10 bitów danych wynosi +-2.

Realizacja kodowania 5b/6b jest nieco bardziej skomplikowana, polega bowiem na takim dopasowaniu '0' i '1' aby różnica pomiędzy ich ilością była równa +-1 (jest to tak na prawdę RD lub odległość Hamminga[6]). Wartość wyjściowa musi być unikalna. Jest to możliwe dzięki dodaniu do zbioru dodatkowego bitu (unikalność mogłaby zostać zachowana bez niego, natomiast nie zostałby spełniony warunek RD = +-1).

Sporą część kodów da się wyliczyć na podstawie zamiany EDCBA -> abcfe (odwrócenie bitów) oraz dodanie wartości '0' lub '1' w zależności od aktualnej wartości RD. Niestety nie dotyczy to wszystkich elementów. Istnieją 32 wartości wraz z przekonwertowanymi odpowiednikami oraz jedna konwersja dla instrukcji kontrolnej.

Konwersja 3b/4b zawiera dwie specyficzne (podkonwersje): podstawową (ang. primary) oraz

[3] HGFEDCBA - liczba zapisana binarnie w postaci 8 bitów, gdzie H jest najbardziej znaczącym bitem a A jest najmniej znaczącym bitem

[4] abcdifhgj - 10 bitowy rezultat kodowania 8b/10b, należy zwrócić uwagę iż kolejność najbardziej znaczącego i najmniej znaczącego bitu jest odwrócona w stosunku do danych wejściowych, oraz i, j, są dodatkowymi dwoma bitami

[5] RD (ang. Running Disparity) - różnica pomiędzy ilością jedynek i zer w ciągu)

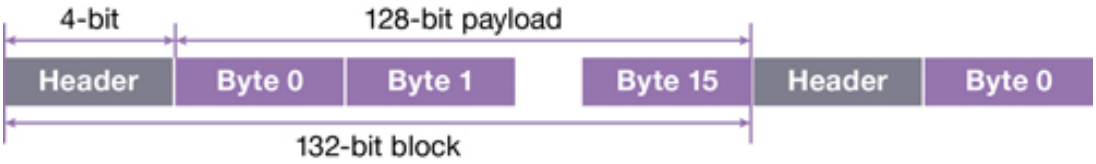
[6] odległość Hamminga - w kontekście projektu XOR dwóch zbiorów

alternatywną (ang. alternative). Dodatkowa alternatywna konfiguracja dla tego zestawu danych została stworzona w celu uniknięcia 5-elementowego ciągu złożonego z samych zer lub jedynek podczas łączenia z kodowaniem 5b/6b, czyli tworzeniem kodu 8b/10b (a w wypadku samego podstawowego zestawu byłoby to o wiele trudniejsze).
Tabela II.1 przedstawia instrukcje kontrolne w postaci zakodowanej w zależności od Running Disparity (RD). Jest to doskonały przykład jak wyglądają tego typu kody.

				RD = -1	RD = +1
	DEC	HEX	HGF EDCBA	abcdei fghj	abcdei fghj
K.28.0	28	1C	000 11100	"001111 0100	"110000 1011
K.28.1	60	3C	"001 11100"	"001111 1001	"110000 0110
K.28.2	92	5C	"010 11100	"001111 0101	"110000 1010
K.28.3	124	7C	"011 11100	"001111 0011	"110000 1100
K.28.4	156	9C	"100 11100	"001111 0010	"110000 1101
K.28.5	188	BC	"101 11100	"001111 1010	"110000 0101
K.28.6	220	DC	"110 11100	"001111 0110	"110000 1001
K.28.7	252	FC	"111 11100	"001111 1000	"110000 0111
K.23.7	247	F7	"111 10111	"111010 1000	"000101 0111
K.27.7	251	FB	"111 11011	"110110 1000	"001001 0111
K.29.7	253	FD	"111 11101	"101110 1000	"010001 0111
K.30.7	254	FE	"111 11110	"011110 1000	"100001 0111

Tab. II.1. *Przykład komunikatów kontrolnych [15]*

Kodowane zaimplementowane w standardzie USB3.1 (128b/132b) różni się nieco od swojego poprzednika. W kodowaniu 128b/132b pierwsze cztery bity zostały przeznaczone na header, a pozostałe 128 bitów na dane. W porównaniu do kodowania 8b/10b, gdzie mieliśmy tak naprawdę utratę 20% możliwości łącza, przez reprezentację każdej porcji 8 bitów - 10 bitowym kodem, to wartość ta spada do 3%. Dodatkowo dzięki pierwszym czterem bitom możliwe jest określenie czy przesyłane są dane do przetworzenia czy wysyłane jest polecenie kontrolne. Dane są obudowywane w pakiety o znanym rozmiarze (zawartym w headerze) o maksymalnej wartości równej 128 bity.



Rys. II.11. *Schemat ramki w kodowaniu 128b/132b [16]*

Sterowniki dla najczęściej używanych urządzeń USB (np. pendrive) instalują się automatycznie na dzisiejszych systemach operacyjnych (ponieważ system operacyjny posiada informacje na temat urządzeń najczęściej użytkowanych), ma to miejsce zaraz po podłączeniu urządzenia do wejścia USB w komputerze osobistym (mechanizm plug-and-play). Celem takiego rozwiązania było jak największe ułatwienie użytkownikowi dostępu do najbardziej spopularyzowanych urządzeń (najczęściej nośników danych, kamer internetowych, etc..). Założeniem projektu było napisanie aplikacji która wymienia dane z urządzeniem niestandardowym (dedykowanym do bardziej złożonych zadań), a co za tym idzie system operacyjny nie był w posiadaniu odpowiednich sterowników dla urządzenia testowego (dokładnie opisanego w rozdziale V). Ten sam przypadek będzie miał miejsce z wykorzystaniem innego urządzenia do testów, lub też, w wypadku wykorzystania komercyjnego, należy zadbać o odpowiednią instalację sterownika. Jeśli chodzi o systemy z rodziny UNIX należy zainstalować bibliotekę (opisaną w rozdziale IV) używając typowego mechanizmu dla systemów UNIXowych[7]. Jest to jedyna czynność jaką należy wykonać przed rozpoczęciem pracy nad implementacją. W przypadku produktów Microsoftu, sprawa przedstawia się nieco inaczej. Wymagana jest instalacja sterownika, który pozwala systemowi na identyfikacje urządzenia. Istotna tutaj jest instalacja sterownika dla każdego nieznanego (nowo podłączonego) przez system urządzenia oddzielne, a dokładniej przy pierwszym użyciu (systemy z rodziny Windows rozpoznają urządzenia połączone interfejsem USB po ich znanej wcześniej VendorId oraz ProductId[8], stad instalacja sterownika odwołuje się do konkretnego podłączonego urządzenia). Po spełnieniu powyższych wymagań możliwe jest pełne wykorzystanie możliwości bibliotek.

Podsumowując, USB należy do rozbudowanych interfejsów poprzez wieloletnie udoskonalanie produktu oraz dążenie do poprawy szybkości przesyłu danych. Dzięki pracy wielu inżynierów oraz udoskonalaniu kodowania znaków, aby rozwiązywać coraz to nowo napotykane problemy, jest to najbardziej powszechny interfejs udostępniony do użytku publicznego. System operacyjny zapewnia szybką instalację sterowników dla podstawowych (najczęściej spotykanych dla użytku domowego) urządzeń. W wypadku bardziej rozbudowanych to programista ma za zadanie zadbać o odpowiednie sterowniki.

[7] w wypadku Ubuntu jest to komenda "sudo apt-get install libusb-1.0-0-dev", gdzie libusb-1.0-0-dev jest nazwą pakietu, zawierającego wszystkie potrzebne elementy biblioteki opisanej w późniejszych rozdziałach

[8] VendorId, ProductId - id specyficzne dla każdego urządzenia, opisane w dalszych rozdziałach

II.3. Testy jednostkowe, komponentowe, automatyczne i wydajnościowe

Tworzenie testów jest nieodłączną częścią tworzenia oprogramowania. Istnieje wiele metodologi stosowanych w tym temacie, a do najbardziej popularnych należy TDD[9] polegająca na stworzeniu przez programistę/testera testu na podstawie wymagań danej funkcjonalności, zanim powstanie do niego implementacja właściwa. Metodologia ta ma swoje plusy jak i minusy. Plusem z pewnością jest weryfikacja funkcjonalności w trakcie pisania kodu produktu (widoczny jest postęp, choć sam w sobie test jeszcze nie przechodzi). Po zakończeniu funkcjonalności możliwy jest pożądany refaktoring kodu, mający na celu zwiększenie przejrzystości dla osób rozwijających projekt dalej. Dzięki zaimplementowanym wcześniej testom istnieje możliwość wstecznego sprawdzania funkcjonalności, innymi słowy czy refaktoring zbytnio nie uszkodził funkcjonalności stworzonego kodu.

Testy dzielą się na kilka grup ze względu na ich funkcjonalność. Podstawowym rodzajem są testy jednostkowe (UT[10]) polegające na testowaniu zachowania poszczególnej funkcji czy też metody. Internet szerzy się od różnego rodzaju frameworków dla UT. Do jednych z najpopularniejszych należy Google Test (jeśli chodzi o C++), który posiada bardzo łatwy do nauczenia się i przystępny framework. Za ich pomocą można testować wynik wyjściowy poszczególnych metod/funkcji, jak również sprawdzać działanie jeśli za argumenty przyjmują wskaźniki. Istnieje również możliwość łatwego zbadania co dana metoda zrobiła z danym obiektem (którego wskaźnik został podany jako argument).

Kolejnym rodzajem testów jakie powinny być przeprowadzane w miarę możliwości są testy komponentowe (SCT[11]). W tym wypadku testowanie odbywa się na zupełnie innych zasadach i nie zawsze jest aplikowalne do każdego projektu. Testy komponentowe należy tworzyć jeżeli projekt/produkt zawiera w sobie więcej niż jeden komponent[12] oraz istnieje zaprojektowana interakcja na interfejsach zdefiniowanych miedzy nimi. Innymi słowy warunkiem koniecznym jest aby te dwa (lub więcej) komponenty komunikowały się ze sobą za pomocą wiadomości. Istotą komponent testów jest to, iż przygotowuje się zbliżone środowisko do docelowego (jeżeli aplikacja działa na hardwarze o określonej architekturze katalogów to należy dostarczyć zbliżoną architekturę do docelowej, często stosuje się do tego kontenery), a następnie uruchamia się tylko i wyłącznie testowany komponent, a jego otoczenie (inne procesy komunikujące się z nim w projekcie) symuluje. Cały test polega na bombardowaniu testowanego komponentu wiadomościami aby wymusić odpowiednie zachowanie. Dla przykładu istnieją 3 komponenty A, B oraz C, testowanym jest komponent B, wiadomo, że w

[9] TDD - ang. Test Driven Development
[10] UT - Unit Tests
[11] SCT - Software Component Tests
[12] tutaj w rozumieniu procesu

docelowym zachowaniu komponent A wysyła do komponentu B wiadomość x, a komponent B po wykonaniu kilku operacji wysyła wiadomość y do komponentu C. Programista testując komponent B powinien mu wstrzyknąć wiadomość x (której struktura w teście jest znana) za pomocą symulowanego komponentu A i sprawdzić czy w takim przypadku symulowany komponent C otrzyma wiadomość y. Jeśli tak się niestanie oznacza to, że jeszcze jakaś część komponentu B nie została zaimplementowana według wymagań, w przeciwnym wypadku test uznał pewną zależność komponentów za poprawną. Popularnymi językami w których istnieje możliwość pisania testów komponentowych są Python oraz ttcn3.

Istnieje również automatyzacja testów. Stosuje się ją w wypadku większych zmian aby mieć pewność iż wprowadzanie nowych funkcjonalności nie naruszyło poprzednich. Automatyzuje się przede wszystkim testy weryfikacyjne (SyVe[13]), które w standardowym (albo w większości procesów) polegają na manualnym testowaniu danych funkcjonalności i weryfikowaniu czy otrzymany został pożądany rezultat. Istotną rolę pełnią również testy integracyjne aplikowalne tylko do większych projektów, w których istotną role odgrywa współdziałanie wielu komponentów, z reguły implementowane przez różne zespoły.

Z punktu widzenia pracy najbardziej istotna jest weryfikacja oprogramowania z początkowymi założeniami (wymaganiami). Należy zadbać o powtarzalność testu i o przejrzystość formatu danych wyjściowych. Testy należy wykonać w środowisku docelowym (z zewnętrznym urządzeniem) aby uniknąć późniejszych wątpliwości. Należy również przygotować środowisko dla automatyzacji. Testy wykonywane są na specyficznym hardwarze, dokładny opis znajduje się w rozdziale V.

[13] SyVe - System Verification

III. Biblioteki udostępniające API do komunikacji z USB

Głównym celem projektu jest osiągniecie jak najszybszej prędkości przesyłu danych. Do realizacji celu konieczne jest użycie odpowiedniej biblioteki do komunikacji z interfejsem USB. W rozdziale przedstawiony został krótki opis najbardziej popularnych bibliotek posiadających interfejs komunikacyjny ze złączami USB.

III.1. Biblioteka libUSB

LibUSB [17] jest biblioteką stworzoną w 2007 roku. Napisana w języku C pozwala na prosty i łatwy dostęp do urządzenia USB. Jest w 100% przeznaczona dla użytku developera. Biblioteka ma za zadanie ułatwić pisanie aplikacji opartych na komunikacji USB z mikrokontrolerem. Jest ona przenośna, a co za tym idzie dostępna na wiele platform (Linux, OS X, Windows, Android, OpenBSD, etc.) wraz z niezmiennym API. Nie są wymagane dodatkowe uprawnienia aby komunikacja z urządzeniem przebiegała poprawnie. Wspiera standardy USB:

- USB1.0
- USB1.1
- USB2.0
- USB3.0

Funkcjonalność biblioteki:

1. wszystkie typy transferu są wspierane (control, bulk, interrupt, isochronous)
2. dwa interfejsy
   (a) synchroniczny (prosty)
   (b) asynchroniczny (bardziej złożony)
3. stosowanie wątków jest bezpieczne
4. lekka biblioteka z prostym API
5. kompatybilna wstecznie (do wersji libUSB-0.1)

III.2. Biblioteka winUSB

Microsoft Windows począwszy od systemu Windows Vista wprowadził nowy zestaw bibliotek umożliwiający developerom korzystnie z portów USB. WinUSB udostępnia proste

API, które pozwala aplikacji na bezpośredni dostęp do portów USB. Został stworzony w gruncie rzeczy dla prostych urządzeń obsługiwanych tylko przez jedną aplikację, takich jak urządzenia do odczytu wskaźników pogodowych czy też innych programów, które potrzebują szybkiego i bezpośredniego dostępu do portu. WinUSB udostępnia API aby odblokować developera przy pracy z portami USB z poziomu user-mode. W Windowsie 7 USB Media Transfer Protocol (MTP) używa winUSB zamiast poprzednio stosowanych rozwiązań kernela (krenel mode filter driver).

Media Transfer Protocol jest rozszerzeniem PTP (Picture Transfer Protocol) i jest protokołem pozwalającym na przesyłanie atomowe plików audio oraz wideo z oraz do urządzenia. PTP początkowo został zaprojektowany do ściągania zdjęć, obrazów z aparatów cyfrowych. Przykładową implementacją może być przypadek przesyłania danych bezpośrednio z cyfrowych urządzeń odtwarzających muzykę oraz pliki video z urządzeń pozwalających na ich odtworzenie.

MTP jest częścią frameworku "Windows Media" blisko związanym z odtwarzaczem Windows Media Player. Systemy Windows począwszy od Windows XP SP2 wspierają MTP. Windows XP wymaga Windows Media Player w wersji 10 lub wyższej, późniejsze wersje systemu wspierają już go domyślnie. Microsoft posiada dodatkowo możliwość zainstalowania MTP na wcześniejszych wersjach systemu ręcznie do wersji Microsoft Windows 98.

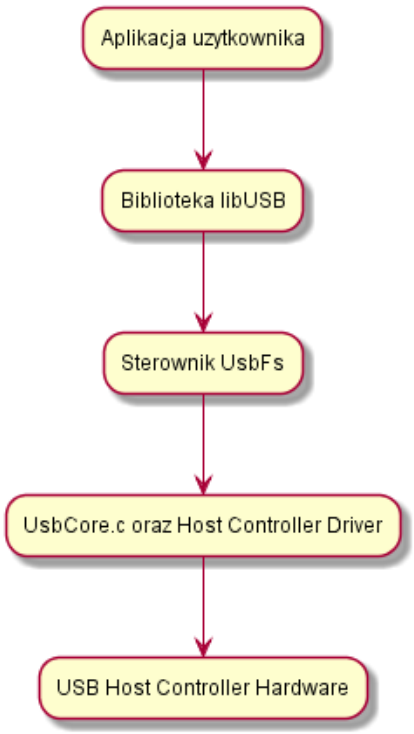
Twórcy standardu USB ustandaryzowali MTP jako pełnoprawną klasę dla urządzeń USB w maju 2008r. Od tamtej pory MTP jest oficjalnym rozszerzeniem PTP i współdzieli ten sam kod klasy. [18, 19, 20, 21]

Podsumowując w projekcie została użyta biblioteka libUSB. Głównym powodem była możliwość kompilacji programów zarówno pod systemami Unix jak i z rodziny Microsoft Windows bez jakichkolwiek (lub znikomych) zmian w kodzie.

IV. Interfejs udostępniony przez libUSB

Zadaniem rozdziału jest opis działania poszczególnych funkcji użytych w projekcie. Jest to konieczne aby przeglądając implementacje projektu być w stanie bezproblemowo zrozumieć działanie poszczególnych części projektu. Rozdział został podzielony na podstawie struktur oraz funkcji, których użycie widoczne jest w implementacji właściwej. [23]

Biblioteka pozwala na łatwy dostęp do poszczególnych funkcjonalności standardów USB. Dzieje się to poprzez udostępnienie API[1], dzięki któremu programista może w pełni wykorzystywać możliwości USB. Poniższy rysunek (Rys. IV.1) przedstawia jak zaprojektowano użycie libUSB pod systemami Linuxowymi. Widać, iż aplikacja użytkownika korzysta tylko i wyłącznie z biblioteki libUSB, natomiast sama biblioteka korzysta ze sterownika UsbFs[2], dzięki czemu zdobywa informacje na temat urządzeń USB. Kolejnym krokiem jest wykorzystanie UsbCore.c oraz sterownika Host Controller. UsbCore.c jest niskopoziomową implementacją standardu USB dla systemów Unixowych, natomiast systemowy sterownik Host Controller pozwala na odblokowanie komunikacji na zewnątrz. To właśnie tutaj zachodzi wydawanie bezpośrednich komend do złącza USB, które jest podłączone bezpośrednio do urządzenia.

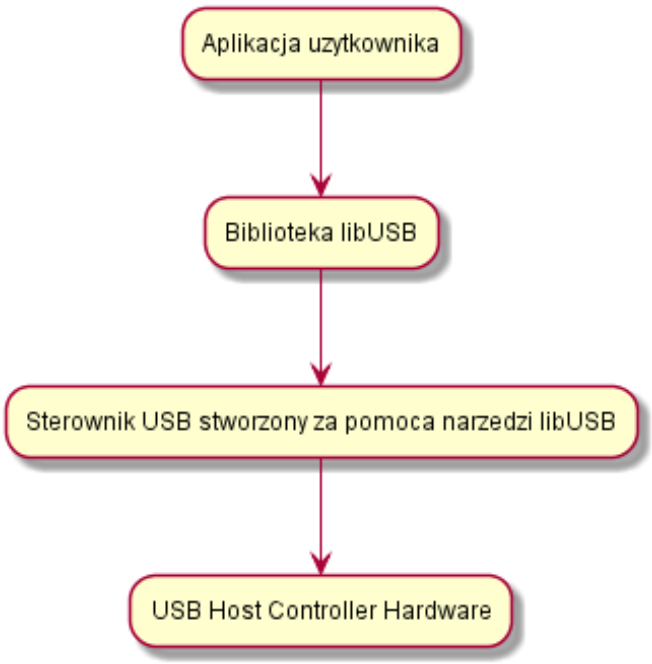


Rys. IV.1. *Schemat działania libUSB pod systemami z rodziny Linux*

[1] API (ang. Application Programming Interface) - interfejs (zazwyczaj zestaw funkcji, struktur) umożliwiający programiście na korzystanie z różnych funkcjonalności biblioteki

[2] UsbFs (ang. Usb filesystem) - wirtualny system plików dostarczający informacji na temat urządzeń USB (podobnie jak /proc na temat procesów

W przypadku działania libUSB dla systemów firmy Microsoft (przedstawione na Rys IV.2) schemat jest nieco uproszczony. Aplikacja korzysta z API udostępnionego przez libUSB (dokładnie z tego samego jak w przypadku działania pod systemem Linux). Natomiast biblioteka libUSB korzysta bezpośrednio ze sterownika zainstalowanego dla konkretnego urządzenia. Pod systemami Windows twórcy biblioteki libUSB stworzyli sterownik, który należy zainstalować oddzielnie dla każdego urządzenia z którym ma przebiegać komunikacja (urządzenie identyfikowane za pomocą vendorId oraz productId - dokładniej opisane w rozdziale II). Poprzez zdanie "za pomocą narzędzi libUSB" (poniższy rys.) rozumiane jest narzędzie pozwalające na łatwą instalacje sterownika dla konkretnego podłączonego urządzenia [22]. Sterownik pozwala na komunikacje ze złączem USB.



Rys. IV.2. *Schemat działania libUSB pod systemami z rodziny Windows*

Przedstawione powyżej schematy ukazują zasadę działania biblioteki pod poszczególnymi systemami operacyjnymi. W wypadku Linux'a korzysta on z systemowych sterowników, natomiast w wypadku systemu Windows wymagana jest dodatkowa instalacja specyficznego sterownika, który pozwala na przetłumaczenie libUSB API na konkretne działania.

IV.1. Charakterystyka wybranych struktur

IV.1.1. libusb_context

Struktura libusb_context reprezentuje sesję libusb. Koncepcja indywidualnych sesji libusb pozwala aby program mógł korzystać z dwóch bibliotek (lub dynamicznie ładować dwa mo-

duły), z których obie niezależnie korzystają z libusb. To zapobiega ingerencji (interferencji) pomiędzy dwoma programami używającymi libusb. Dla przykładu libusb_set_debug() nie zaingeruje w działanie innego programu korzystającego z libusb, natomiast libusb_exit() nie wyczyści pamięci używanej przez inny program libusb. Sesje tworzone są za pomocą libusb_init() oraz czyszczone za pomocą libusb_exit(). Jeśli zagwarantowane jest to, że dana aplikacja jest jedyną która korzysta z libusb, twórca jej nie musi przejmować się kontekstami (strukturą libusb_context), wystarczy aby przekazywał do wszystkich funkcji, gdzie struktura jest wymagana wartość NULL. Jest to równoważne z użyciem domyślnego kontekstu.

IV.1.2. libusb_device_handle

Struktura reprezentująca uchwyt do urządzenia USB. Jest to nieprzejrzysty typ, użycie jest możliwe tylko za pomocą wskaźnika, zazwyczaj dostarczanego za pomocą libusb_open(). Uchwyt do urządzenia USB jest używany do wykonywania operacji wejścia/wyjścia. Po zakończeniu wszystkich działań na uchwycie należy wywołać libusb_close().

IV.2. Charakterystyka użytych funkcji

IV.2.1. libusb_init

Funkcja ta odpowiada za inicjalizację biblioteki. Musi on zostać wywołana przed jakiejkolwiek inną funkcją z biblioteki libUsb. Jeśli w argumencie nie zostanie dostarczony żaden wskaźnik (wyjściowy) kontekstu, zostanie stworzony kontekst domyślny. W przypadku jeśli domyślny kontekst już istnieje zostanie on ponownie użyty (bez inicjalizacji). Funkcja zwraca wartość 0 w wypadku powodzenia, w przeciwnej sytuacji zwraca kod błędu.

IV.2.2. libusb_open_device_with_vid_pid

Wygodna funkcja służąca do odszukania konkretnego urządzenia na podstawie jego vendorId oraz productId (są to parametry charakteryzujące każde urządzenie). Używana jest w przypadkach kiedy z góry jest znane vendorId oraz productId. Najczęstsze zastosowania znajdują przy pisaniu aplikacji w celu przetestowaniu jakiejś określoną funkcjonalność. Funkcja pozwala uniknąć wywołania libusb_get_device_list() i dbania o odpowiednie czyszczenie pamięci po liście. Pierwszym jej parametrem jest kontekst uzyskany za pomocą libusb_init(). Natomiast kolejnymi parametrami są vendorId oraz productId. Funkcja zwraca uchwyt do znalezionego urządzenia, lub NULL w wypadku kiedy nie może znaleźć pożądanego urządzenia (o podanym productId oraz vendorId) lub błędu.

IV.2.3. libusb_kernel_driver_active

Funkcja sprawdzająca czy sterownik jądra kernela jest aktywny na interfejsie. W wypadku kiedy sterownik jądra jest aktywny nie możliwe jest zgłoszenie użycia interfejsu, a co za tym idzie libUSB nie może wykonać operacji wejścia/wyjścia. Funkcjonalność nie jest dostępna w systemie Windows. Do jej zadań należy przekazać uchwyt do urządzenia oraz numer interfejsu. Na wyjściu zwraca ona wartość 0, jeśli żaden sterownik jądra nie jest aktywny, wartość 1 w wypadku jeśli istnieje aktywny sterownik jądra.
Do zwracanych błędów miedzy innymi należą: LIBUSB_ERROR_NO_DEVICE jeśli urządzenie zostanie odłączone, LIBUSB_ERROR_NOT_SUPPORTED dla platform, gdzie funkcjonalność nie jest wspierana lub gdy występuje inny rodzaj błędu (nie wymieniony wyżej).

IV.2.4. libusb_detach_kernel_driver

Dzięki funkcji możliwe jest odłączenie sterownika jądra kernela od interfejsu. Sukces operacji umożliwi zgłoszenie użycia interfejsu i wykonanie operacji wejścia/wyjścia. Funkcjonalność nie jest dostępna dla systemu Windows. Parametrami są uchwyt do urządzenia oraz numer interfejsu. Funkcja zwraca wartość 0 w momencie powodzenia, natomiast w przeciwnym wypadku LIBUSB_ERROR_NOT_FOUND jeśli żaden sterownik nie był aktywny, LIBUSB_ERROR_INVALID_PARAM jeśli interfejs nie istnieje, LIBUSB_ERROR_NO_DEVICE jeśli urządzenie zostało odłączone, LIBUSB_ERROR_NOT_SUPPORTED dla platform, gdzie funkcjonalność nie jest wspierana lub gdy występuje inny rodzaj błędu (nie wymieniony w tej sekcji)

IV.2.5. libusb_claim_interface

Dzięki funkcji libusb_claim_interface() możliwe jest zgłoszenie użycia danego interfejsu dla danego urządzenia. Wywołanie funkcji jest wymagane przed wykonaniem operacji wejścia-/wyjścia dla dowolnego punktu końcowego interfejsu. Jest dozwolone wywołanie funkcji dla interfejsu już wcześniej zgłoszonego, w tym wypadku zostanie zwrócona wartość 0 bez wykonywania żadnych operacji. W wypadku jeśli zmienna auto_detach_kernel_driver jest ustawiona na wartość 1 dla danego urządzenia (zmienna ustawiana za pomocą funkcji: libusb_set_auto_detach_kernel_driver()) sterownik jądra zostanie odłączony (jeśli to konieczne), w przypadku niepowodzenia zostanie zwrócony błąd odłączenia. Sama procedura wewnątrz funkcji nie jest skomplikowana, nie wymaga wysyłania czegokolwiek po magistrali. Są to proste instrukcje mówiące systemowi operacyjnemu iż aplikacja chce korzystać z danego interfejsu. Nie jest to funkcja blokująca. Parametrami są uchwyt do urządzenia oraz numer interfejsu zgłaszanego. Wartość 0 zostaje zwrócona w wypadku powodzenia operacji. W wypadku niepowodzenia kody błędu t.j. LIBUSB_ERROR_NOT_FOUND jeśli podany

interfejs nie istnieje, LIBUSB_ERROR_BUSY jeśli inny program lub sterownik zarezerwował dany interfejs, LIBUSB_ERROR_NO_DEVICE jeśli urządzenie zostało odłączone oraz inne.

IV.2.6. libusb_bulk_transfer

Dzięki funkcji możliwy jest przesył większej grupy danych. Jest to jedna z najważniejszych funkcji użytych w projekcie. Kierunek jest określony na podstawie bitów kierunkowych punktu końcowego interfejsu. Dla odczytu, jeden z parametrów określa ilość danych jaka jest spodziewana przy odczycie. Jeżeli odebrana zostanie mniejsza ilość danych niż oczekiwana, funkcja po prostu zwróci te dane wraz z dodatkowym parametrem określającym ich ilość. Istotne przy odczycie jest to aby sprawdzić, czy ilość oczekiwanych danych jest taka jak odczytana. W wypadku zapisu również należy sprawdzić czy ilość danych wysłanych pokrywa się z ilością danych skierowaną do wysyłki. Wskazana jest również weryfikacja ilości danych wysłanych/odebranych w wypadku wystąpienia timeoutu (funkcja zwróci kod błędu określający timeout). LibUSB może podzielić wysyłane dane na mniejsze części i timeout może wystąpić po wysłaniu kilku z nich. Ważne jest to, że nie oznacza to iż nic nie zostało wysłane/odebrane, dlatego należy sprawdzić ilość elementów wysłanych/odebranych i dostosować odpowiednio kolejne kroki. Funkcja przyjmuje następujące parametry: uchwyt urządzenia z którym aplikacja będzie się komunikować, adres punktu końcowego interfejsu po którym będzie odbywała się komunikacja, wskaźnik do pamięci danych która ma zostać przetransferowana (w wypadku zapisu) lub odebrana (w wypadku odczytu), ilość danych do wysłania (w przypadku zapisu) lub oczekiwana ilość danych do odebrania (w przypadku odczytu), ilość danych przetransferowanych (w obu przypadkach), maksymalna długość czasu na wykonanie operacji, dla nieograniczonego należy użyć wartości równej 0. W przypadku poprawności działania funkcja zwraca wartość 0 oraz ilość przetransferowanych danych przekazanych do funkcji za pomocą wskaźnika. W przeciwnym wypadku funkcja zwraca: LIBUSB_ERROR_TIMEOUT jeśli transfer przekroczył określony czas, LIBUSB_ERROR_PIPE jeśli wystąpił błąd związany z punktem końcowym, LIBUSB_ERROR_OVERFLOW jeśli urządzenie wysłało więcej danych niż przewidziane w buforze, LIBUSB_ERROR_NO_DEVICE jeśli urządzenie zostało odłączone oraz inne.

IV.2.7. libusb_release_interface

Funkcja zwalnia rezerwacje wcześniej zgłoszonego interfejsu za pomocą funkcji opisanej wyżej (libusb_claim_interface). Zwolnienie wszystkich interfejsów jest wymagane przed zamknięciem urządzenia. Nie jest to blokująca funkcja. W wypadku jeśli zmienna auto_detach _kernel_driver jest ustawiona na wartość 1 dla danego urządzenia (zmienna ustawiana za pomocą funkcji: libusb_set_auto_detach_kernel_driver()) sterownik jądra zostanie ponow-

nie podłączony zaraz po zwolnieniu interfejsu. Parametrami są: uchwyt urządzenia oraz numer interfejsu dla niego poprzednio zarezerwowanego.
Metoda zwraca 0 gdy wszystkie operacje się powiodą. W przeciwnym wypadku zwraca: LIBUSB_ERROR_NOT_FOUND, jeśli interfejs nie został poprzednio zarezerwowany (zgłoszony do użycia) lub LIBUSB_ERROR_NO_DEVICE jeśli urządzenie zostało odłączone oraz inne.

IV.2.8. libusb_close

Zwalnia uchwyt do urządzenia. Wymagane jest, aby była wołana na wszystkich używanych poprzednio uchwytach. Jej zadaniem jest zniszczenie referencji stworzonej za pomocą libusb_open() dla danego urządzenia. Jest to funkcja nie blokująca. Parametrem jest uchwyt przeznaczony do zamknięcia.

IV.2.9. libusb_exit

Funkcja zamyka dostęp do biblioteki. Powinna być wołana po zamknięciu wszystkich otwartych urządzeń, ale przed zakończeniem działania programu. Parametrem jest kontekst, który ma zostać zamknięty, w wypadku wartości NULL wybierany jest domyślny.

IV.2.10. libusb_alloc_transfer

Funkcja przygotowuje transfer z wyspecyfikowaną ilością izochronicznych deskryptorów pakietów. Zwraca ona uchwyt do zainicjalizowanego transferu. Kiedy wszystkie operacje zostaną na nim wykonane należy wywołać libusb_free_transfer(). Transfer przygotowywany dla nieizochronicznego punktu końcowego należy wywoływać z wartością zero jako ilość izochronicznych pakietów. Natomiast dla transferów izochronicznych wymagane jest podanie prawidłowej wartości deskryptorów pakietów jakie mają zostać zalokowane w pamięci. Warto wspomnieć, że zwracany transfer nie jest domyślnie zainicjalizowany jako izochroniczny, dodatkowo wymagane jest ustawienie pola libusb_iso_packets oraz type. Alokowanie transferu jako izochroniczny (z podana ilością deskryptorów pakietów do zalokowania), a następnie używanie go jako nieizochroniczny jest w 100% bezpieczne, ale tylko pod warunkiem, że pole num_iso_packets jest ustawione na zero oraz pole type jest ustawione prawidłowo. Funkcja jako prametr przyjmuje ilość izochronicznych deskryptorów pakietów. Zwracaną wartością jest zalokowany transfer lub NULL w wypadku błędu.

IV.2.11. libusb_fill_bulk_transfer

Funkcja pozwala na łatwe przygotowanie struktury libusb_transfer na transfer masowy.
Parametrami są:

- uchwyt do ustawianego transferu,
- uchwyt do urządzenia, dla którego ustawiany jest transfer,
- adres punktu końcowego gdzie dane mają zostać wysłane,
- bufor danych do wysłania/odebrania,
- długość (wielkość) wysyłanych/odbieranych danych,
- wskaźnik do funkcji, która ma się wywołać po zakończeniu transferu (callback),
- dodatkowe dane, które programista może opcjonalnie wysłać do funkcji wywołanej po zakończeniu transferu,
- czas oczekiwania w milisekundach na zakończenie transferu.

IV.2.12. libusb_submit_transfer

Funkcja wykonuje podany (ustawiony) transfer. Wykonuje ona operacje na interfejsie po czym bezzwłocznie kończy działanie. Jedynym parametrem jaki przyjmuje funkcja jest wskaźnik do transferu jaki ma zostać wykonany.
W wypadku kiedy wszystko się powiedzie funkcja zwraca wartość równą 0, w pozostałych przypadkach zwraca kod błędu tj. LIBUSB_ERROR_NO_DEVICE w wypadku kiedy urządzenie nie jest podłączone, LIBUSB_ERROR_BUSY w przypadku jeśli akcja została już wykonana, LIBUSB_ERROR_NOT_SUPPORTED jeśli flagi transferu (ustawienia) nie są wspierane przez system operacyjny oraz inne.

IV.2.13. libusb_free_transfer

Funkcja odpowiada za zwolnienie pamięci zajętą przez strukturę libusb_transfer Powinna być zawołana dla wszystkich transferów zaalokowanych przez libus_alloc_transfer(). Jeśli dodatkowo flaga LIBUSB_TRANSFER_FREE_BUFFER jest ustawiona oraz bufor transferu jest nie zerowy, pamięć po nim zostanie zwolniona za pomocą standardowej funkcji alokującej (np. free()). Dozwolone jest wywołanie z parametrem równym NULL, w takim przypadku nie zostanie zwrócony błąd (ale również zwolnienie zasobów nie zostanie wykonane). Niedozwolone jest zwalnianie pamięci po niezakończonym (aktywnym) jeszcze transferze, czyli takim, który wystartował, ale jeszcze nie został zawołany callback do niego (lub timeout). Parametrem funkcji jest wskaźnik do transferu przeznaczony do zwolnienia.

Podsumowując, rozdział ten opisuje najważniejsze z funkcji biblioteki libUSB, które zostały użyte podczas implementacji projektu. Opis każdej z nich zawiera nazwę, parametry, zwracaną wartość w przypadku błędu oraz krótkie wyjaśnienie jej działania.

V. Urządzenie testowe

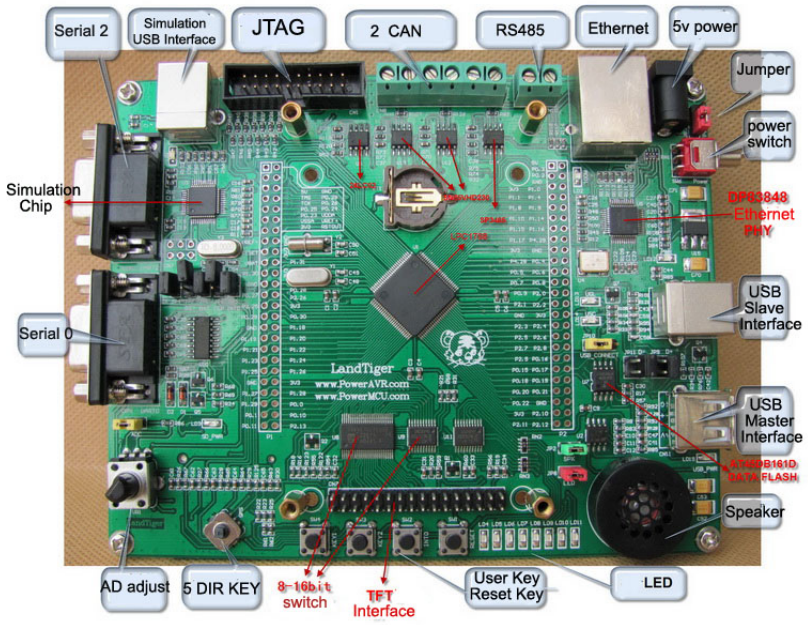
Aby zrealizować projekt konieczne było użycie dodatkowego hardwaru, którego zadanie polegało na odbieraniu danych po przeciwnej stronie jak oraz ich przesyłanie w przeciwnym kierunku. Inwestygacja rozpoczęła się od bardzo prostych urządzeń takich jak raspberry pi B+, a zakończyła się na mikrokontrolerze LandTiger.

Rys. V.1. *Mikrokontroler LandTiger wraz z wyświetlaczem [24]*

Mikrokontroler LandTiger oparty na LPC1768 został wyprodukowany przez firmę PowerMCU i można go zakupić od wielu dostawców na eBay lub innych serwisach świadczących usługi zakupów przez internet. Średni koszt waha się w okolicach $70 za płytkę wraz z wyświetlaczem LCD 3,2 cala o rozdzielczości 320x240 pikseli, z zasilaczem oraz zestawem kabli. [24]
Funkcjonalności:

- 2 porty RS232, jeden z nich wspiera ISP (In-system Programming),
- 2 interfejsy magistrali CAN (Controller Area Network),
- interfejs RS485,
- interfejs Ethernetowy RJ45-10/100M,
- przetwornik cyfrowo-analogowy (DAC) wraz wmontowanym głośnikiem (wyjściem interfejsu) oraz sterownikiem dźwięku (LM386),

- przetwornik analogowo-cyfrowy (ADC) wraz z wbudowanym potencjometrem (wejściem interfejsu),
- Kolorowy 3,2 cala (lub 2,8 cala) dotykowy wyświetlacz LCD o rozdzielczości 320x240 pikseli,
- interfejs USB2.0 (USB Host oraz USB Device),
- interfejs kard SD/MMC,
- interfejs I2C połączony z 2Kbit pamięcią EEPROM,
- interfejs SPI połączony z 16Mbit pamięcią flash,
- 2 user keys, 2 function keys,
- 8 diód typu LED,
- pięciokierunkowy joystick,
- wsparcie dla pobierania ISP,
- pobieranie z użyciem JTAG, interfejs dla debugowania,
- zintegrowany emulator kompilacji JLINK - wspiera możliwość debugownia online (po kablu USB podłączonym do PC) dla środowisk deweloperskich tj. KEIL, IAR, CooCox i innych,
- dodatkowe 5V port zasilający (możliwe jest też za pomocą portu USB.



Rys. V.2. *Mikrokontroler LandTiger wraz z opisem poszczególnych elemetnów [25]*

LandTiger jest oparty na LPC1768. Wbudowany hardware wspiera ISP aby umożliwić załadowanie kodu (z użyciem bin2hex oraz flashmagic). Alternatywą jest to, że kod może zostać załadowany za pomocą emulatora JLINK JTAG/SWD lub za pomocą zewnętrznego urządzenia JTAG.

Port COM1 (UART0) wspiera komunikację z PC w obie strony. Wszelkie funkcje portu USB

są wspierane z minimalnymi zmianami w oprogramowaniu. Podobnie jest z Ethernetem, z niewielkimi zmianami w oficjalnym kodzie dla LPC1768, kod jest w stanie się uruchomić na LandTigerze.

Wyświetlacz LCD jest oparty na kontrolerze SSD1289. Wyświetlacz może zostać odłączony od płyty. Używa on 8-bitowej magistrali $P2_0..P2_7$. Kontroler ekranu dotykowego jest dostarczony razem z modułem wyświetlacza. Interfejs pomiędzy ekranem dotykowym a LPC1768 jest możliwy dzięki SPI.

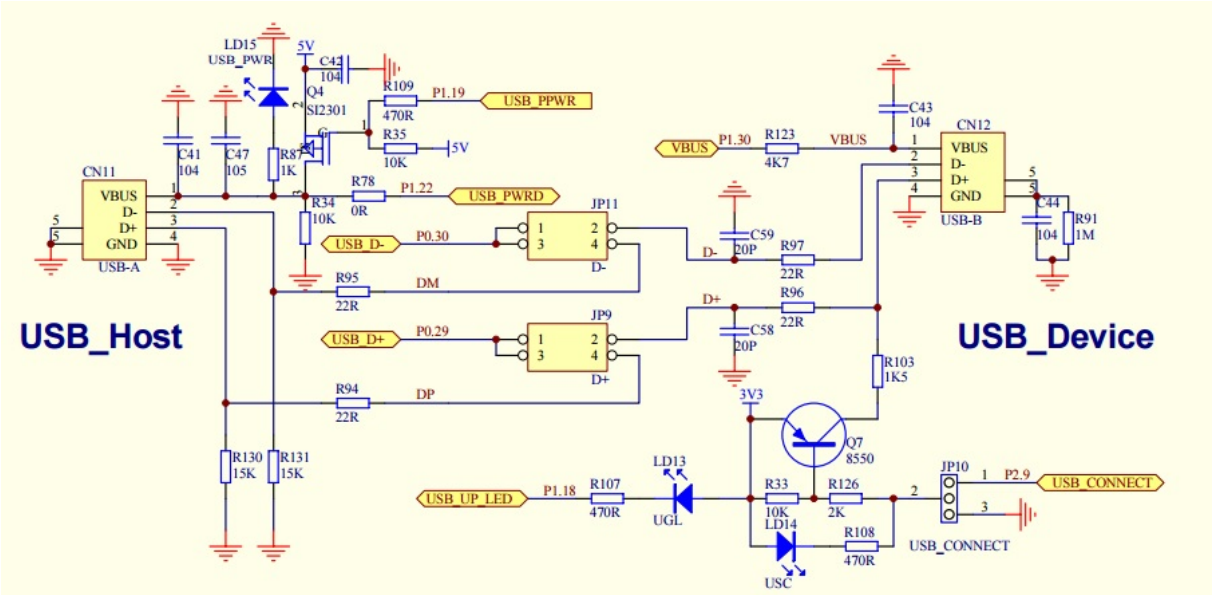
Główne różnice pomiędzy LandTigerem a LPC1768:

- płyta po podłączeniu do PC nie pokazuje się jako zewnętrzne urządzenie magazynujące,
- aby ściągnąć nowe pliki binarne należy użyć ISP lub JTAG,
- brak wsparcia dla serialowego portu po linku USB, należy używać RS232 lub portu USB,
- brak wsparcia dla logicznego systemu plików,
- brak wsparcia dla 4 diód typu LED (istnieje możliwość użycia inncyh).

Rozdział ukazuje, że Mikrokontroler LandTiger jest rozbudowanym urządzeniem zawierającym szereg różnego rodzaju funkcjonalności. Doskonale nadaje się jako urządzenie dla osób, które nie miały zbyt dużej styczności z programowaniem embedded, a jego przejrzyste API oraz dokumentacja pozwala na szybkie wdrożenie się w środowisko. [24]

VI. Implementacja

Implementacja opiera się na wykorzystaniu prostych podstawowych funkcji z biblioteki libUsb. Rozwiązania zostały obudowane w odpowiednie klasy wraz z zastosowaniem polimorfizmu, aby uzyskać możliwość łatwego dodawania nowych udoskonaleń. Przed rozpoczęciem projektowania klas należało zaznajomić się jak przedstawione jest USB na schemacie ideowym płytki LandTiger (Rys. VI.1).



Rys. VI.1. *Schemat ideowy USB na płytce LandTiger[24]*

W projekcie wykorzystywano jedno z przedstawionych na rysunku złączy. Był nim USB_Device, a co za tym idzie wszystkie zworki musiały zostać odpowiednio przestawione aby umożliwić jego działanie (JP9, JP10, JP11). Na schemacie widoczne jest, iż jest to złącze typu B (grubsze). Składa się z czterech pinów:

- Pin 1 - VBUS, odpowiada za dostarczenie zasilania +5V,
- Pin 2 - terminal do przesyłania odwróconego sygnału (D-),
- Pin 3 - terminal do przesyłania sygnału (D+)
- GND - uziemienie

Istotne jest to aby długości D+ oraz D- były takie same, wtedy sygnał pokonuje dokładnie takie same odległości i zapewniona zostaje integralność. W wypadku kiedy odległości są różne mogą wystąpić różnego rodzaju błędu związane z wyeliminowaniem zakłóceń (eliminacja zakłóceń opisana dokładniej w rozdziale II). Na schemacie różnica jest niezauważalna. Dodatkowo na poszczególnych liniach użyto filtry dolnoprzepustowe aby wyeliminować z widma sygnału częstotliwości powyżej częstotliwości Nyquista, czyli częstotliwości z jaką (w wypadku elektroniki) możliwy jest odczyt sygnału bez błędów, takich jak nakładania się składowych widmowych sygnału o wyższych częstotliwościach (niż częstotliwość Nyquista) na

składowe o niższych częstotliwościach.

Po zapoznaniu się ze schematem ideowym, należało przystąpić do realizacji projektu. Diagram klas użytych w projekcie znajduje się na rysunku VI.2. Doskonale pokazuje to łatwość rozszerzalności o nowe funkcjonalności aplikacji. Jeżeli w przyszłości uda się opracować nowy sposób przesyłania danych, można rozszerzyć klasę Mode o tą funkcjonalność. Opis poszczególnych klas wraz z użytymi metodami znajduje się w poniższych podrozdziałach. Klasa abstrakcyjna Mode udostępnia bardzo podstawowe metody służące do inicjalizacji oraz czyszczenia zasobów. W wypadku tworzenia własnego trybu oraz konieczności innej inicjalizacji (lub zwalniania zasobów) należy dostarczyć własną implementacje tych metod. Konieczne jest dostarczenie implementacji testu. Dokładny opis znajduje się poniżej.

Rys. VI.2. *Diagram klas projektu*

VI.1. Klasa Mode

Klasa Mode jest nadzbiorem funkcjonalności. Obudowuje ona podstawowe funkcjonalności takie jak:

- inicjalizacja

  pobieranie kontekstu,

  pobieranie uchwytu urządzenia,

  rejestracja interfejsu,

- oczyszczanie pamięci.

Definicja zawiera również czysto wirtualną metodę doTest(), której odpowiednie implementacje w obiektach dziedziczących dostarcza funkcjonalności testom.

```
class Mode
{
public:
  Mode(int bufforSize, unsigned count, int vid, int pid, bool printOnlyResult) :
    _bufforSize(bufforSize), _count(count), _vid(vid), _pid(pid), _printOnlyResult(
    printOnlyResult)
  {
    _debugPrinter.set(!printOnlyResult);
  }
  virtual ~Mode();
  virtual int generateSymulatedData(unsigned char*, const int);
  virtual void printFinalInformation();
  virtual void initProcedures();
  virtual void getContext();
  virtual void getDeviceHandle();
  virtual void proceedWithInitUsb();
  virtual void closeLibUsb();
  virtual int doTest() = 0;

protected:
  int _bufforSize;
  unsigned _count;
  int _vid;
  int _pid;
  double _timeResult;
  DebugPrinter _debugPrinter;
  bool _printOnlyResult;
  libusb_context* _ctx;
  libusb_device_handle* _dev_handle;

};
```

Listing VI.1. *Deklaracja klasy Mode*

VI.1.1. Metoda getContext

W listingu VI.2 przedstawione zostało ciało metody getContext() klasy Mode.
Metoda nie przyjmuje żadnego argumentu, natomiast inicjalizuje kontekst i zapamiętuje go jako składową klasy. W rozdziale IV wspomniane zostało iż możliwe jest używanie kontekstu domyślnego, wtedy zamiast użycia libusb_init() należałoby przypisać wskaźnikowi do kon-

tekstu wartość równą NULL.
Aby używać domyślnego kontekstu należy mieć pewność iż aplikacja/wątek jest jedynym użytkownikiem libUSB.
W wypadku niepoprawnego działania wyrzucany jest błąd (ważne aby był przechwycony w funkcji main).

```
void Mode::getContext()
{
  int r = libusb_init(&_ctx);
  if(r < 0) {
    throw std::runtime_error("Init Context error");

  }
}
```

Listing VI.2. *Metoda Mode::getContext()*

VI.1.2. Metoda getDeviceHandler

Metoda nie przyjmuje żadnego argumentu. Jej zadaniem jest ustawienie uchwytu do urządzenia na podstawie jego vendorId oraz productId. W wypadku niepowodzenia zostanie zgłoszony odpowiedni błąd (np. braku podłączonego i uruchomionego mikrokontrolera). Ciało funkcji zostało przedstawione w Listingu VI.3

```
void Mode::getDeviceHandle()
{
  _dev_handle = libusb_open_device_with_vid_pid(_ctx, _vid, _pid);
  if(_dev_handle == NULL)
    throw std::runtime_error("Cannot open device!");
  else
    _debugPrinter << "Device Opened\n";
}
```

Listing VI.3. *Metoda Mode::getDeviceHandler()*

VI.1.3. Metoda proceedWithInitLibUsb

Metoda ta odpowiedzialna jest za dokończenie procedur inicjalizacyjnych. Ciało metody zostało przedstawione w Listingu VI.4. Metoda sprawdza dodatkowo czy sterownik jądra kernela jest aktywny, dla platformy Windows funkcja zwróci wartość błędu, która będzie równa wartości enumerycznej LIBUSB_ERROR_NOT_SUPPORTED (!= 1) i zachowa się analogicznie jak w wypadku nieaktywnego sterownika w systemie UNIX (warunek będzie niespełniony).

Kolejnym krokiem jest rezerwacja przez program konkretnego interfejsu za pomocą funkcji libusb_claim_interface.
W wypadku niepoprawnego działania metoda wyrzuci runtime_error, który należy obsłużyć w funkcji main().

```
void Mode::proceedWithInitUsb()
{
  if(libusb_kernel_driver_active(_dev_handle, 0) == 1) { //find out if kernel driver
    is attached
    _debugPrinter << "Kernel Driver Active\n";
    if(libusb_detach_kernel_driver(_dev_handle, 0) == 0) //detach it
      _debugPrinter << "Kernel Driver Detached!\n";
  }
  int status = libusb_claim_interface(_dev_handle, 1);
  if(status < 0)
  {
    throw std::runtime_error("Cannot Claim Interface");
  }
  _debugPrinter << "Claimed Interface\n";
}
```

Listing VI.4. *Metoda Mode::proceedWithInitLibUsb()*

VI.1.4. Metoda initProcedures()

Metoda przedstawiona w listingu VI.5 odpowiada za wywołanie procedur inicjalizacji w odpowiedniej kolejności.

```
void Mode::initProcedures()
{
  getContext();
  getDeviceHandle();
  proceedWithInitUsb();
}
```

Listing VI.5. *Metoda Mode::initProcedures()*

VI.1.5. Metoda closeLibUsb

Metoda przedstawiona w Listingu VI.6 odpowiada za zwolnienie interfesju oraz zasobów uprzednio zajętych na czas testu.

```
void Mode::closeLibUsb()
{
  int status = libusb_release_interface(_dev_handle, 1);
```

```
  if(status != 0) {
    throw std::runtime_error("Cannot Relase Interface");
  }
  _debugPrinter << "Released Interface\n";
  libusb_close(_dev_handle);
  libusb_exit(_ctx);
}
```

Listing VI.6. *Metoda Mode::closeLibUsb()*

VI.1.6. Metoda printFinalInformation

Metoda mająca za zadanie wypisanie ostatecznych wyników. Wyniki mogą zostać przedstawione jako czytelna zwykła informacja dla użytkownika lub w postaci kolumnowej (używanej przy tworzeniu wykresów).

```
void Mode::printFinalInformation()
{
  _debugPrinter << "Sending of: " << _bufforSize * _count << "Bytes using bufferSize="
     << _bufforSize << " takes " << _timeResult << "s.\n";
  if(_printOnlyResult)
  {
    unsigned allSendReceivedData = _bufforSize * _count;
    double sendingTime = _timeResult/2;
    double receivingTime = _timeResult/2;

    printf("%d\t%u\t%u\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\t%.2f\n", _bufforSize, _count,
    allSendReceivedData, _timeResult, sendingTime, receivingTime,
              allSendReceivedData/_timeResult, allSendReceivedData/sendingTime,
    allSendReceivedData/receivingTime);
  }
}
```

Listing VI.7. *Metoda Mode::printFinalInformation()*

VI.2. Klasa SynchMode

Klasa SynchMode jest odpowiedzialna za konfigurację USB dla przesyłu synchronicznego.

```
class SynchMode : public Mode
{
public:
  SynchMode(int bufforSize, unsigned count, int vid, int pid, bool printOnlyResult);
```

```
  virtual ~SynchMode();
  virtual int doTest() override;

private:

};
```

Listing VI.8. *Deklaracja klasy SynchMode*

Używa ona wszystkich domyślnych ustawień (nie przeciąża żadnej metody wirtualnej), oczywiście metoda doTest() wymaga implementacji.

VI.2.1. Metoda doTest

Metoda jest odpowiedzialna za wykonanie podstawowego pomiaru czasu przepływu danych w obie strony, pomiędzy PC a mikrokontrolerem. Została zaprojektowana w taki sposób aby konkretny bufor danych został wysłany oraz odebrany określoną ilość razy i zwrócony przedział czasowy w jakim udało się to uzyskać. Powodem wysyłania/odbierania danych określoną ilość razy jest fakt sporych ograniczeń jeśli chodzi o chip wbudowany w płytę LandTiger. Fragment metody został przedstawiony w listingu VI.9.

```
int SynchMode::doTest()
{
//...
    int sendStatus = libusb_bulk_transfer(_dev_handle, (2 | LIBUSB_ENDPOINT_OUT),
    data_out, _bufforSize, &howManyBytesIsSend, 0);
//...
    int readStatus = libusb_bulk_transfer(_dev_handle, (2 | LIBUSB_ENDPOINT_IN),
    data_in, _bufforSize * sizeof(unsigned char), &howManyBytesReceived, 0);
//...
}
```

Listing VI.9. *Fragment metody SynchMode::doTest()*

VI.3. Klasa AsynchMode

Klasa AsynchMode odpowiada za konfigurację USB dla przesyłu asynchronicznego.

```
class AsynchMode : public Mode
{
public:
  AsynchMode(int bufforSize, unsigned count, int vid, int pid, bool printOnlyResult);
  virtual ~AsynchMode();
  virtual int doTest() override;
```

```
  virtual void initProcedures() override;
  void closeLibUsb() override;
private:
  pthread_mutex_t _sender_lock;
  pthread_mutex_t _receiver_lock;
  pthread_cond_t _sender_cond;
  pthread_cond_t _receiver_cond;

  libusb_transfer* _senderTransfer;
  libusb_transfer* _receiverTransfer;
  pthread_t _receiverThread;

};
```

Listing VI.10. *Deklaracja klasy AsynchMode*

Klasa zawiera wzbogaconą inicjalizację o alokację transferów (wysyłającego oraz odbierającego) oraz inicjalizację mutexów. Istotnym elementem jest przedstawiony w listingu VI.11 dodatkowy namespace ułatwiający obsługę callbacków (w listingu przedstawiono jego strukturę). Znajduje się w nim zarówno obsługa wątku nasłuchującego jak i obsługa poszczególnych zdarzeń.

```
namespace ThreadHelper
{
  struct TransferStatus
  {
  //...
  };
  static void LIBUSB_CALL cb_read(struct libusb_transfer *transfer)
  {
    //..
  }
  static void LIBUSB_CALL cb_send(struct libusb_transfer *transfer)
  {
    //...
  }
  void* receiverThread(void* arg)
  {
  //..
  }
}
```

Listing VI.11. *Struktura przestrzeni nazw oraz odpowiednich struktur do poprawnej obslugi wątków*

VI.3.1. Metoda initProcedures oraz closeLibUsb

W wypadku asynchronicznego przesyłania danych należy wykonać kilka dodatkowych operacji inicjalizacyjnych. Są to alokacje transferów oraz inicjalizacja mutexów. Całość widoczna w listingu VI.12.

```
void AsynchMode::initProcedures()
{
  Mode::initProcedures();
  _senderTransfer = libusb_alloc_transfer(0);
  _receiverTransfer = libusb_alloc_transfer(0);
  if(_senderTransfer == NULL || _receiverTransfer == NULL)
  {
    throw std::runtime_error("Transfer allocation Error");
  }

  if(pthread_mutex_init(&_sender_lock, NULL) != 0 || pthread_mutex_init(&
    _receiver_lock, NULL) != 0)
  {
    throw std::runtime_error("Mutex Init Failed!");
  }

}
```

Listing VI.12. *Metoda AsynchMode::initProcedures*

Analogicznie sytuacja przedstawia się w wypadku zwalniania zasobów. Dodatkową czynnością jest usunięcie mutexów oraz zwolnienie transferów. Całość dostępna w listingu VI.13.

```
void AsynchMode::closeLibUsb()
{
  pthread_mutex_destroy(&_sender_lock);
  pthread_mutex_destroy(&_receiver_lock);
  libusb_free_transfer(_senderTransfer);
  libusb_free_transfer(_receiverTransfer);
  Mode::closeLibUsb();
}
```

Listing VI.13. *Metoda AsynchMode::closeLibUsb*

VI.3.2. Metoda doTest

Podobnie jak w wypadku synchronicznego mechanizmu tak i teraz należy dostarczyć implementację metody doTest(). Natomiast jej działanie jest odmienne, ponieważ polega na wystartowaniu drugiego wątku nasłuchującego odpowiedzi. Całość synchronizowana jest za pomocą mutexów (listing VI.14 oraz VI.11).

```
int AsynchMode::doTest()
{
//...
  libusb_fill_bulk_transfer(_senderTransfer, _dev_handle, (2 | LIBUSB_ENDPOINT_OUT),
    data_out, _bufforSize, ThreadHelper::cb_send, &transferStatus, 0);
  libusb_fill_bulk_transfer(_receiverTransfer, _dev_handle, (2 | LIBUSB_ENDPOINT_IN),
    data_in, _bufforSize, ThreadHelper::cb_read, &transferStatus, 0);
//..
  if(pthread_create(&_receiverThread, NULL, ThreadHelper::receiverThread, &
    transferStatus) != 0)
  {
    throw std::runtime_error("Error creating listener thread.");
  }
//...
    pthread_mutex_lock(&_sender_lock);
    while(transferStatus.waitForSender){
      pthread_cond_wait(&_sender_cond, &_sender_lock);
    }
    transferStatus.waitForSender = 1;
    pthread_mutex_unlock(&_sender_lock);
    transferStatus.particularSendComplete = 0;
    libusb_submit_transfer(_senderTransfer);
  //...
}
```

Listing VI.14. *Metoda AsynchMode::doTest()*

VI.4. Korzystanie z poszczególnych interfejsów

Kod przedstawiony w Listingu VI.15 doskonale ukazuje prostotę korzystania z libUSB. Użytkownik zobligowany jest do wprowadzenia wielkości bufora danych oraz liczbę określającą ilość powtórzeń z jaką testowa aplikacja wyśle wygenerowane dane (o rozmiarze podanego bufora) do mikrokontrolera. Następnie zostaje wykonana inicjalizacja z użyciem wyżej wymienionych interfejsów (synchronicznego lub asynchronicznego na podstawie parametru).

```
int main(int argc, char* argv[])
{
  //..
  try
  {
    std::unique_ptr<Mode> mode;
```

```
    if(selectedMode == 'A')
      mode.reset(new AsynchMode(bufforSize, count, LAND_TIGER_VID, LAND_TIGER_PID,
    printOnlyResult));
    else
      mode.reset(new SynchMode(bufforSize, count, LAND_TIGER_VID, LAND_TIGER_PID,
    printOnlyResult));
    mode->initProcedures();
    mode->doTest();
    mode->printFinalInformation();
    mode->closeLibUsb();
  } catch (std::runtime_error& e)
  {
    std::cout << e.what() << std::endl;
  }

  return 0;
}
```

Listing VI.15. *Przykład użycia interfejsu synchronicznego lub asynchronicznego w zależności od parametryzacji*

VI.4.1. Parametryzacja

Aby ułatwić uruchamianie poszczególnych testów wprowadzono możliwość sparametryzowania wejścia aplikacji:

```
use: libusb [S/A] <bufforSize> <fullDataSize> optional:<printOnlyResult>
```

Listing VI.16. *uruchomienie testu*

- S/A - należy wybrać czy chcemy wykonać test w trybie Synchronicznym czy Asynchronicznym,
- bufforSize - rozmiar bufforu danych jaki chcemy przeznaczyć dla jednej paczki danych [B],
- fullDataSize - całkowity rozmiar danych do wysłania [B],
- printOnlyResult - opcjonalny parametr w wypadku ustawienia na wartość '1' pozwala na łatwe zebranie wyników bez zbędnych informacyjnych wiadomości (same dane).

VI.5. Kod działający po stronie mikrokontrolera

Aby test przebiegł prawidłowo konieczne było zaimplementowanie prostej funkcjonalności po stronie mikrokontrolera, mającej za zadanie krótkie potwierdzenie odebrania wiadomości (jako reakcja na zdarzenie) lub wykonanie akcji na kontrolerze. Poniższy listing

zawiera przykładowy kod wysyłający jako potwierdzenie otrzymane dane (niestety aby nie wystąpił błąd w wypadku większej ilości przysłanych danych, wysyłana jest zawartość ograniczona do 64 B). [26, 27, 28]

```
void USB_Receiver_Sender()
{
//...
  while(1)
  {
    CDC_OutBufAvailChar (&numAvailByte);
    if (numAvailByte > 0)
    {
        numBytesToRead = numAvailByte > MAX_TEMP_BUFFOR ? MAX_TEMP_BUFFOR :
    numAvailByte;
        numBytesRead = CDC_RdOutBuf (&serBuf[0], &numBytesToRead);
        USB_WriteEP (CDC_DEP_IN, (unsigned char *)&serBuf[0], numBytesRead);
    }
  }

}
```

Listing VI.17. *Funkcja USB_Receiver_Sender*

VI.6. Implementacja pomocnicza

Do implementacji pomocniczej należy klasa ułatwiająca wywoływanie aplikacji celem zebrania wyników. Jest ona zaprojektowana jako prosta, posiadająca przeładowany operator dodawania do strumienia danych. W wypadku tego projektu, dla normalnego działania aplikacji operator pozwala na wypisanie dowolnych danych, ale w wypadku włączenia zbierania danych w postaci kolumnowej ta klasa nie pozwoli na to.

```
class DebugPrinter
{
public:
  DebugPrinter() : _enabled(true)
  {
  }
  DebugPrinter(bool enabled) : _enabled(enabled)
  {
  }

  DebugPrinter& operator<<(const char ss[])
  {
    if(_enabled)
```

```
      std::cout << ss;
    return *this;
  }
  DebugPrinter& operator<<(const int& d)
  {
    if(_enabled)
      std::cout << d;
    return *this;
  }
  void set(bool enabled)
  {
    _enabled = enabled;
  }

private:
  bool _enabled;
};
```

Listing VI.18. *Klasa DebugPrinter*

W wypadku wywołania aplikacji:

```
use: libusb [S/A] <bufforSize> <fullDataSize> optional:<printOnlyResult>
```

Listing VI.19. *Uruchomienie testu*

bez ostatniego argumentu, w wypadku wykonania tego fragmentu kodu:

```
debugPrinter << "It is impossible to send " << fullDataSize << "B using " <<
    bufforSize << "B, to simplify application work sending " << (++count) * bufforSize
     << "B\n";
```

Listing VI.20. *Wypisywanie w zależności od argumentu wywołania aplikacji*

aplikacja wypisze powyższy tekst, natomiast w wypadku wywołania aplikacji wraz z ostatnim opcjonalnym argumentem <printOnlyResult>=1 spowoduje, że przeładowany operator klasy wspomnianej w listingu VI.18 zablokuje wypisywanie tekstu. Klasa została stworzona tylko po to aby ułatwić przyszłe zbieranie wyników.

VII. Testy wydajności oraz ich rezultaty

Oprogramowanie zostało poddane dogłębnym testom systemowym, mającym na celu zweryfikowanie poprawności działania aplikacji oraz jej zgodności z wymaganiami. Wykonywano je na każdym etapie programowania aby uniknąć jakichkolwiek nieumyślnych interakcji w rezultaty działających uprzednio funkcjonalności. Ważne było aby środowisko testowe było znane. W przypadku tego projektu jest to mikrokontroler LandTiger opisany dokładniej w rozdziale V oraz komputer osobisty o parametrach przedstawionych w tabeli VII.1.

Typ procesora	DualCore Intel Core i3 350M, 2266 MHz (17 x 133)
Pamięć	6GB
Dysk fizyczny	WDC WD3200BPVT-80ZEST0 (298 GB)
USB	2.0

Tab. VII.1. *Podstawowe parametry komputera na którym wykonywane zostały testy*

Istotnym elementem wymagań było to aby aplikacja była wieloplatformowa, włączając przede wszystkim system Windows oraz systemy z rodziny Linux. Testy zostały wykonane na systemie Windows 8.1 Pro oraz na Linux Ubuntu 14.04. Analizując poniższe wyniki testów należy mięć na uwadze poniższe fakty i założenia:

1. Początkowym celem było uzyskanie prędkości nie mniejszej niż 140 Mbit/s co daje 17,5 MB/s w obu kierunkach.

2. Zarówno komputer osobisty jak i płytka LandTiger posiada wbudowane złącze żeńskie standardu USB2.0 (komputer złącze typu A, natomiast płytka złącze typu B).

3. Jak się okazało podczas tworzenia projektu płytka LandTiger, pomimo wlutowanego złącza USB2.0, nie jest w stanie obsłużyć tego standardu. Spowodowane jest to tym iż chip na płycie wspiera jedynie standard USB1.1.

4. warunkiem koniecznym udanego testu jest niestety wsparcie minimum USB2.0.

VII.1. Testy jednostkowe

Testy jednostkowe miały na celu sprawdzenie poprawności działania poszczególnych metod oraz klas projektu. Dodatkowo chroniły proces tworzenia nowych funkcjonalności aby nie ingerowały w działanie uprzednio napisanych. Testy jednostkowe tworzone były równolegle z programowaniem poszczególnych funkcjonalności dzięki czemu udało się zadbać o całkowitą spójność projektu.

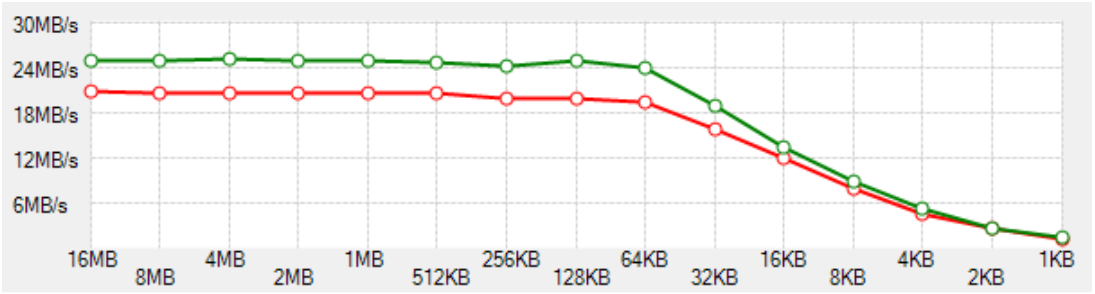
VII.2. Testy akceptacyjne i regresyjne

Testy akceptacyjne mają za zadanie sprawdzenie czy dana aplikacja spełnia wymagania i warunki oraz potwierdzić czy w testowym środowisku aplikacja działa poprawnie. Były one wykonywane równolegle z procesem tworzenia oprogramowania dzięki czemu udało się zidentyfikować problemy w locie. Zostały wykonane na zasadzie black box testing. Metodologia ta polega na tym, iż tester testuje konkretną, zaimplementowaną funkcjonalność bez wglądu w kod produkcyjny, opierając się tylko na wymaganiach danej funkcjonalności. Jest to o tyle słuszne, w wypadkach kiedy testy te wykonywane są przez inną osobę, niż osobę która implementowała funkcjonalność, ponieważ programista może podświadomie wymusić tylko te zachowania aplikacji, które zostały zaimplementowane przez niego, natomiast osoba opierająca się tylko na wymaganiach sprawdzi całość.

VII.3. Zestawienie wyników

Zebrane wyniki możemy skategoryzować według: ilości przesłanych danych, szybkości przesyłu danych z komputera osobistego do kontrolera, szybkości przesyłu danych z kontrolera do komputera osobistego, szybkości przesyłu w obie strony. Kluczowym elementem jest tutaj wielkość buffora, czyli ilości danych możliwych do wysłania/odebrania w jednym tiku zegara.

Według założeń USB 2.0 jest w stanie obsłużyć oczekiwane 140Mbit/s (17,5 MB/s). Wykres przedstawiony na Rys. VII.1 doskonale to obrazuje.



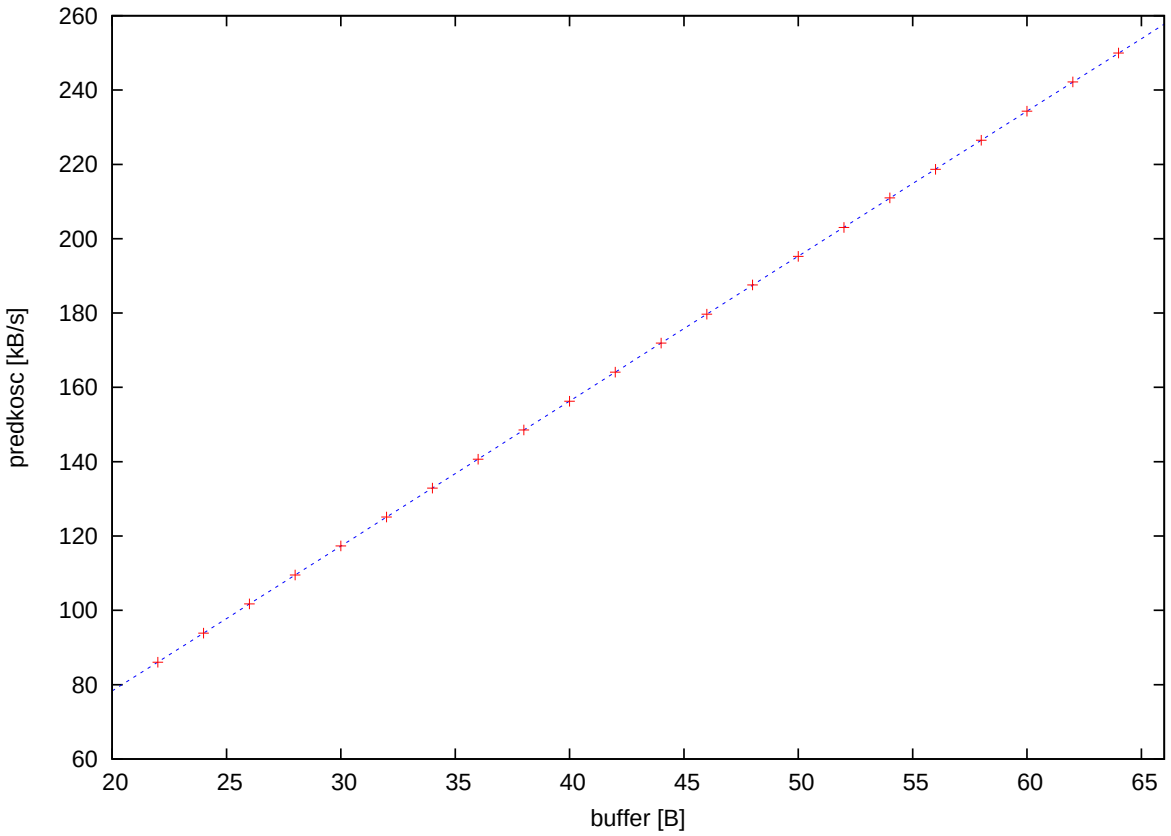
Rys. VII.1. *Zależność prędkości przesyłu danych od wielkości bufora*

Rysunek VII.1 przedstawia wykres prędkości wysyłania oraz odbioru danych w zależności od bufora danych (bufor maleje)[1]. Na wykresie zielona linia przedstawia odczyt danych z użyciem interfejsu USB2.0 natomiast czerwona przedstawia prędkość wysyłania danych. Wybrane wyniki w formie tekstowej są częścią załącznika B niniejszej pracy. Etapem godnym uwagi jest zachowanie wykresu w momencie gdy buffor danych wynosi 64KB, jest to

[1] dane uzyskane dzięki programowi USB Flash Benchmark[29]

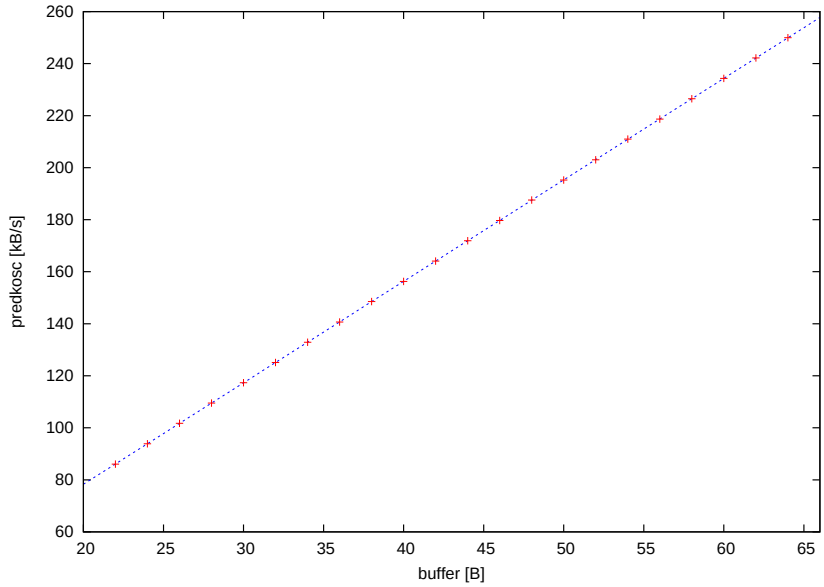
ostatnia stabilna wartość buffora, która umożliwiałaby spełnienie wymagań projektu. W wypadku wartości 32KB widoczny jest jeszcze spełnienie warunku ale tylko dla danych wysyłanych (zielona linia). Pozostałe wartości buffora nie spełniają warunków projektu.

Podczas testów akceptacyjnych równolegle z developmentem okazało się iż płytka użyta w projekcie a dokładniej opisana w rozdziale V wspiera na swoim chipie jedynie standard USB1.1 pomimo wlutowanego złącza USB2.0. Niestety w wypadku USB1.1 jest to 64B. Większość wyników została wygenerowana za pomocą programu, który nie pozwalał na przekroczenie dopuszczalnego przez LandTiger'a bufforu (tak jak w przypadku Rys. VII.2 oraz innych)



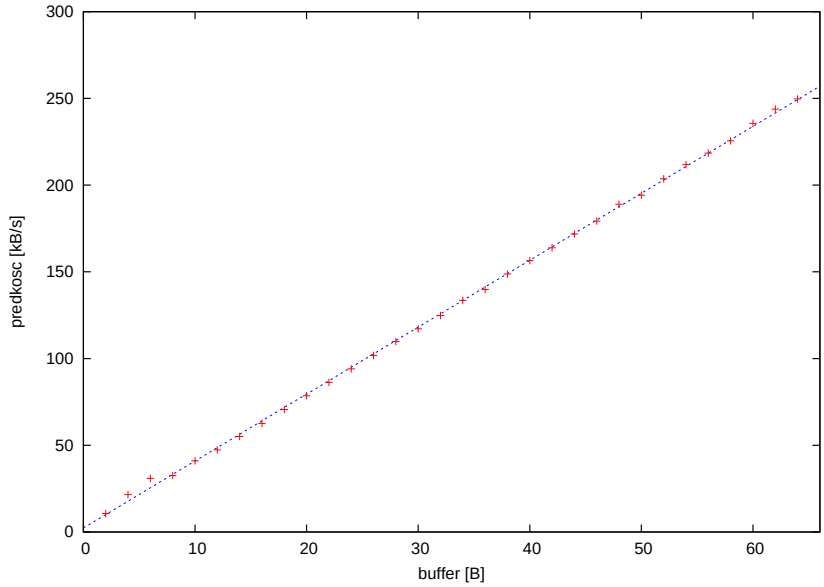
Rys. VII.2. *Wykres ilustrujący prędkość wysyłania danych w zależności od buffora (tryb synchroniczny)*

Na Rys. VII.2 widoczny jest wyraźny wzrost szybkości wysyłania danych wraz ze zwiększeniem bufora. Jest to doświadczenie wykonane na stosunkowo małym buforze, spowodowane jest to ograniczeniami płytki LandTiger. Można zaobserwować liniową zależność pomiędzy zobrazowanymi wielkościami.



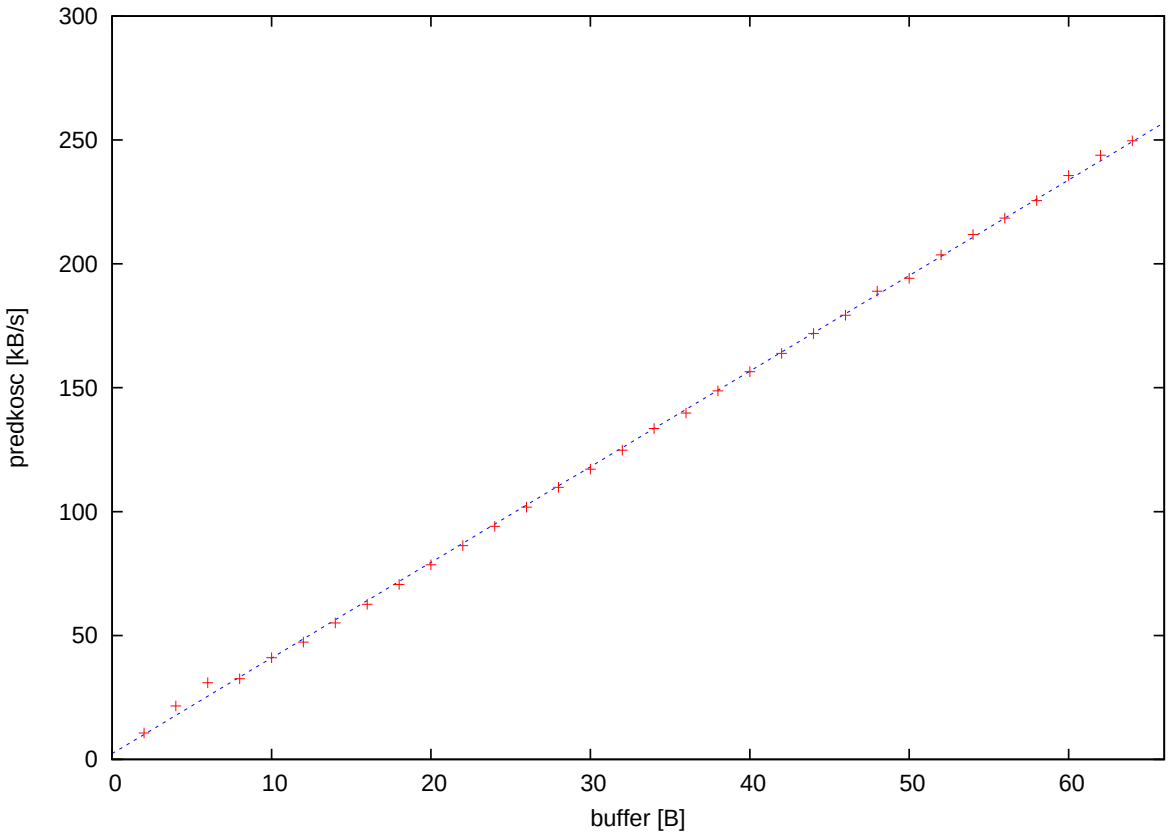
Rys. VII.3. *Wykres ilustrujący prędkość odbierania danych w zależności od buffora (tryb synchroniczny)*

Analogiczny wykres widoczny jest dla prędkości odbierania danych (Rys. VII.3), różnice są niezauważalne przy tak małej ilości danych.



Rys. VII.4. *Wykres ilustrujący prędkość wysłania danych w zależności od buffora (tryb synchroniczny)*

W wypadku wysyłania nieco mniejszej próbki danych prędkość obrazuje się jako nieco mniej stabilna. Całość widoczna jest na Rys. VII.4.
Podobnie jak w poprzednim przypadku (dla większej ilości danych), wykres prędkości odbierania danych pokrywa się z wykresem prędkości wysyłania danych (Rys. VII.5).

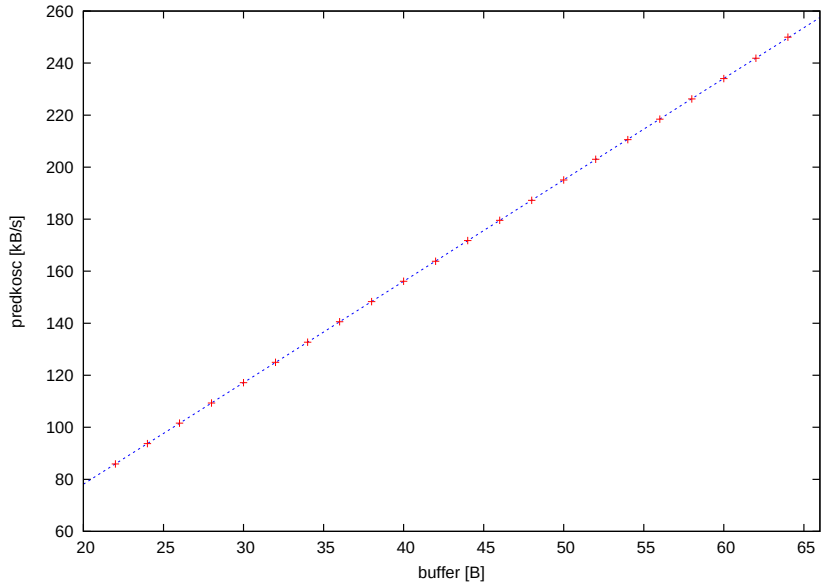


Rys. VII.5. *Wykres ilustrujący prędkość odbierania danych w zależności od buffora dla 10 MB (tryb synchroniczny)*

Wszystkie powyższe wykresy zostały wygenerowane z użyciem mechanizmu synchronicznego. Oznacza to, że obsługa wysyłania oraz odbierania danych zrealizowana została za pomocą wbudowanych bibliotek, co z punktu widzenia kodu aplikacji realizuje taka sama paczka danych wysłanych oraz odebranych. Podsumowując powyższe wyniki można uznać metodę synchroniczną za nieskuteczną dla standardu USB1.1. Oczekiwane wyniki nie zostały w tym wypadku uzyskane. Mechanizmem blokującym jest wielkość bufora danych przy wysyłaniu lub odbieraniu danych (64B). Jest to zdecydowanie za mało aby ta metoda w tym wypadku była skuteczna ale porównując otrzymane wyniki z teoretycznymi (Rys. VII.1 oraz dodatek B) otrzymanymi z użyciem USB benchmark, gdzie najmniejsza wielkość bufora wynosiła 1kB oraz wartość otrzymana oscylowała w okolicach 1,26MB-1,36MB można założyć iż wartości otrzymane z użyciem metody synchronicznej są jak najbardziej poprawne, a ograniczeniem, które nie pozwala na uzyskanie oczekiwanych wartości jest jedynie konieczność użycia standardu USB1.1.

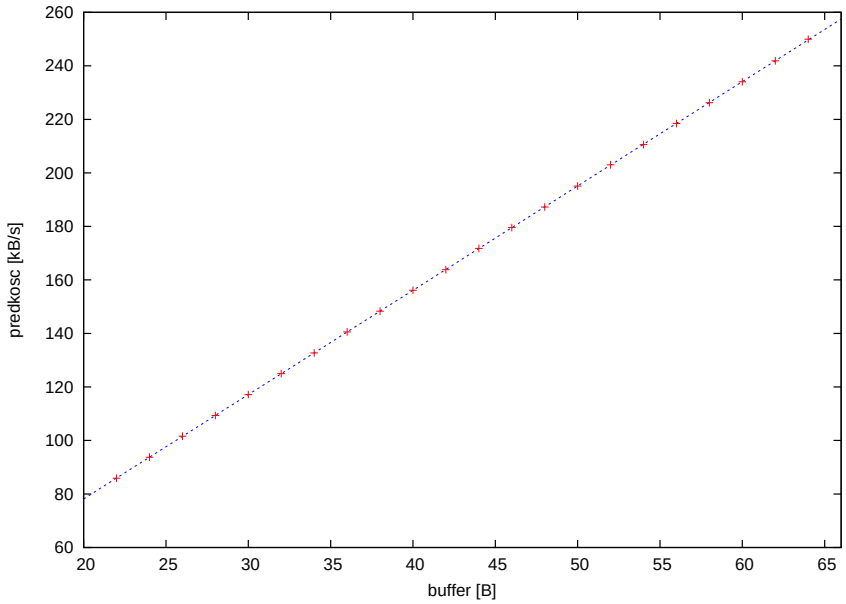
Zupełnie innym podejściem charakteryzuje się metoda asynchroniczna w której korzysta się z nieblokujących metod. Istotne jest, aby zauważyć, że o synchronizacje dba programista (ponieważ po łączu w jednym czasie mogą przebiegać dane tylko w jedną stronę). Na podstawie przeprowadzonego doświadczenia można stwierdzić, że implementacja była du-

żo bardziej kosztowna i wymagała przeznaczenia więcej czasu aby zadbać o poprawną synchronizacje danych. Niemniej to ona doprowadziła do większej kontroli nad zarządzaniem przełączania wysyłania oraz odbierania danych. W wypadku przesyłanych danych z wykorzystaniem LandTiger'a, a co za tym idzie wykorzystaniem jedynie dostępnego rozmiaru buforu danych wielkości 64B, zmiany są praktycznie niezauważalne.



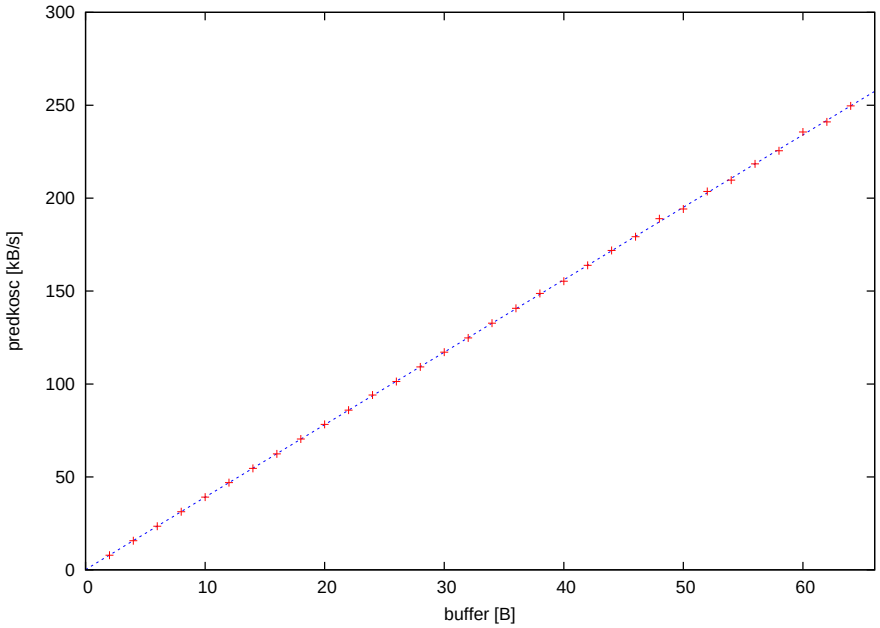
Rys. VII.6. *Wykres ilustrujący prędkość wysyłania danych w zależności od buffora (tryb asynchroniczny)*

Jak widac na Rys. VII.6 zależność liniowa również występuje dla tej ilości danych przy przesyle asynchronicznym.



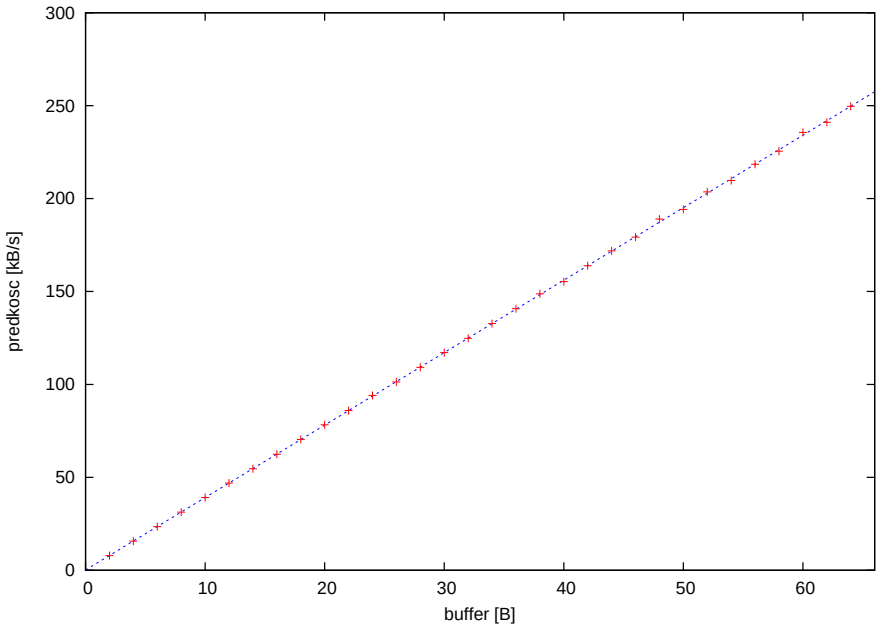
Rys. VII.7. *Wykres ilustrujący prędkość odbierania danych w zależności od buffora dla 102 MB (tryb Asynchroniczny*

Analogiczna sytuacja występuje w wypadku odbierania danych metodą asynchroniczną. Wykres jest liniowy jak jego odpowiednik z metody synchronicznej (Rys. VII.3) a wyniki są bardzo zbliżone do siebie.



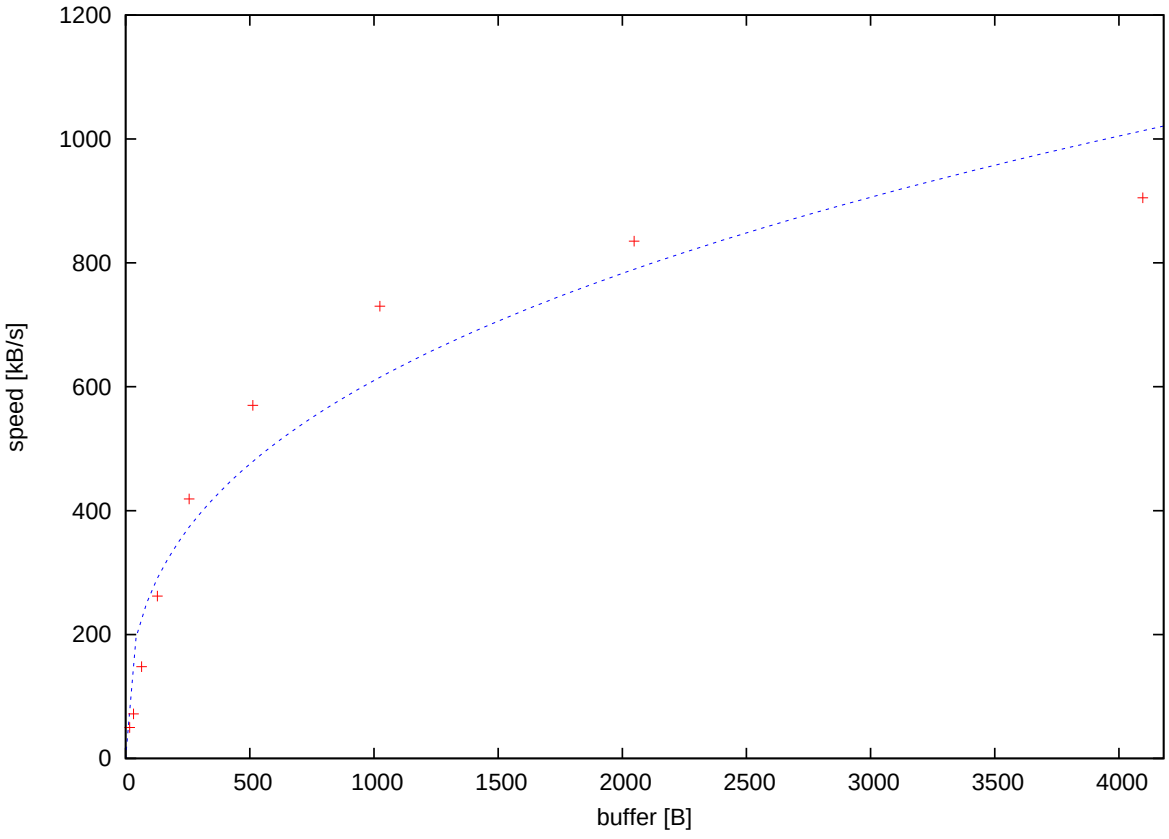
Rys. VII.8. *Wykres ilustrujący prędkość wysyłania danych w zależności od buffora (tryb asynchroniczny)*

Jeśli przeprowadzimy testy dla mniejszej ilości danych to zauważalny jest brak liniowości takiego wykresu (zarówno dla Rys. VII.8 jak i Rys. VII.9).



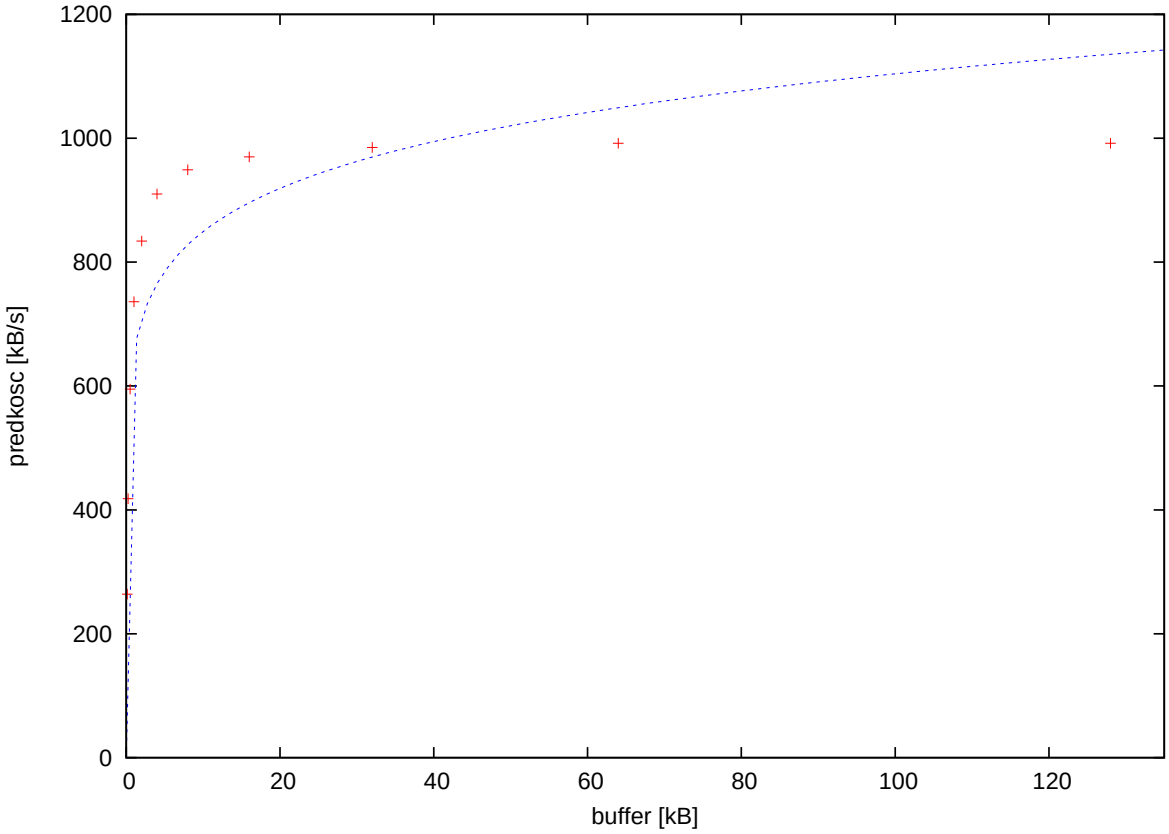
Rys. VII.9. *Wykres ilustrujący prędkość odbierania danych w zależności od buffora (tryb asynchroniczny)*

Wysyłanie metodą Asynchroniczną również nie przyniosła oczekiwanych rezultatów. Porównując je do wartości zasymulowanych na początku rozdziału z użyciem USB benchmark, pozostawiają wiele do życzenia. Jednak jeśli ponowimy zestawienie wyników z teoretycznymi (podobnie jak to zostało to wykonane w wypadku metody Synchronicznej) to zauważymy pewną analogię iż gdyby istniała możliwość wykorzystania bufora większego niż dopuszczalny przez standard USB1.1, wyniki nie odbiegałyby od teoretycznych. Podsumowując, użycie bufora dopuszczalnego przez mikrokontroler LandTiger (czyli bufora nie większego niż 64B) nie pozwala na uzyskanie wyników zbliżonych do wartości pożądanej (140 MB/s) bez względu na to, czy korzystano z metody synchronicznej czy też asynchronicznej. W obu przypadkach dla bufora o wielkości 64B udało się uzyskać maksymalnie 250 kB/s. Wyniki teoretyczne (dodatek B) przedstawiają dla bufora wielkości 1kB wyniki rzędu 1,26MB/s - 1,36 MB/s. Oznacza to, że przy założeniach otrzymanej dotąd liniowości, prędkość ta zostałaby osiągnięta, w wypadku możliwości użycia bufora wielkości 1kB przez aplikację.
Dodatkowo została przeprowadzona symulacja z uwzględnieniem większego bufora danych, jednak w tym przypadku nie udało się uzyskać prawidłowego odbioru danych po stronie kontrolera. Oznacza to, że te dane mogą zostać potraktowane jedynie jako przypuszczenia jak wyglądałby wykres gdyby dane były przetworzone prawidłowo. Poniższe (Rys. VII.10 oraz Rys. VII.11 wykresy przedstawiają wygenerowane wyniki.



Rys. VII.10. *Wykres ilustrujący zmianę prędkości przesyłania danych z użyciem bufora z przedziału 16 B - 4 kB*

Powyższy wykres obrazuje zmianę prędkości wysyłania przy założeniu poprawnego odbioru danych po stronie kontrolera. Zależność liniowa całkowicie zaniknęła. Wartości rozmiaru bufora danych jest podwajana z każdym zgięciem krzywej (początkowa wartość 16 B, końcowa 4096 B = 4 kB), a więc rozbieżność przesyłu danych jest duża. Warto zauważyć iż pomimo dość dużego bufora danych prędkość nie przekroczyła 1 MB/s (max wartość 905,702 kB/s).



Rys. VII.11. *Wykres ilustrujący zmianę prędkości przesyłania danych z użyciem bufora z przedziału 128B - 128kB*

Powyższy wykres jest rozwinięciem poprzednika. Obrazuje zmianę prędkości na podstawie zasymulowanych danych oraz przy założeniach poprawnego odebrania danych przez mikrokontroler. Podstawową różnicą jest wielkość użytego bufora w tym przypadku. Został w nim użyty bufor o wartości minimalnej równej 128 B oraz maksymalnej równej 131072 B (128 kB). Bufor w każdym kroku był podwajany. Początkowo wartość uzyskiwanej prędkości wzrasta gwałtownie w każdym kroku, natomiast powyżej wartości 8 kB nie są już tak gwałtowne a wręcz przechodzi w coraz bardziej poziomy wzrost i dąży do wartości 1 MB/s.

Podsumowując całościowo wszystkie użyte metody w projekcie (Synchroniczna, Asynchroniczna) oraz zwykłą symulacje przesyłu większej ilości danych (pomimo braku możliwej obslugi po stronie mikrokontrolera) można założyć, iż w wypadku realizacji projektu z mikrokontrolerem obsługującym standard nowszy niż USB1.1 rezultaty oczekiwane zostałyby osiągnięte. Dodatkowo zestawiając metodę synchroniczną oraz asynchroniczną z wynikami

teoretycznymi można śmiało potwierdzić, że jest to możliwe (zakładając iż trend wykresów nie ulegnie wielkiej zmianie). Należy mieć również na uwadze iż, mogą istnieć wszelkiego rodzaju problemy i zakłócenia trudne do namierzenia na styku złącza USB2.0 -> kabla USB2.0 -> złącza USB1.1 i na odwrót. Potwierdza to jedynie poprzednie założenia. Ostatnie dwa wykresy przedstawiają symulacje wysyłania danych z użyciem większego bufora danych ale bez poprawnej obsługi ich odbioru po stronie mikrokontrolera (ponieważ nie jest to możliwe z powodu ograniczeń płytki). Widoczne jest na nich, iż prędkość rośnie lecz zmienia trend z czysto liniowego w typowo logarytmiczny. Powodem są prawdopodobne zakłócenia wspomniane wcześniej, niemożliwość przetworzenia danych po stronie płytki, a co za tym aplikacja nie wie kiedy wysłanie partii danych się zakończyła i kiedy powinna rozpocząć wysyłanie kolejnej oraz inne obostrzenia USB1.1. Zestawiając do testów urządzenie obsługujące standard USB2.0, brak obsługi otrzymanych danych nie miałby miejsca co zapewniłoby płynność w przesyłaniu większej ilości danych i zapewne wykresy oparte na podobnych wartościach przedstawiałyby dane zgodne z teoretycznymi.

VIII. Podsumowanie

Celem projektu było przygotowanie oprogramowania umożliwiającego wysyłanie i odbieranie danych z wykorzystaniem złącza uniwersalnej magistrali szeregowej, czyli USB. Aby zrealizować projekt należało poznać standard USB i zrozumieć jego działanie, jak i innych, mniej lub bardziej zaawansowanych portów szeregowych. Istotne było zrozumienie różnic pomiędzy standardami wydanymi do dnia dzisiejszego jako podstawa teoretyczna do zrealizowania projektu. Projekt posiada zintegrowany system testowania funkcjonalności, aby zapobiec negatywnej ingerencji w istniejące już funkcjonalności.

Najważniejszym krokiem podczas tworzenia projektu był wybór urządzenia testowego, czyli urządzenia na którym wykonywane były najważniejsze testy akceptacyjne. Wybór padł na mikrokontroler LandTiger, który został dokładniej opisany w rozdziale V. Aplikacja została zaprojektowana w prosty i rozszerzalny sposób. Głównym punktem projektu jest klasa abstrakcyjna pełniąca podstawowe funkcje oraz posiadająca domyślne implementacje poszczególnych metod. Po tej klasie zostały podziedziczone dwie inne wprowadzające własne wersje implementacji testu, jak i w wypadku bardziej rozbudowanej klasy odpowiadającej za asynchroniczne przesyłanie danych, również inicjalizacja biblioteki przebiega inaczej. Dodatkowo w przyszłości istnieje łatwa możliwość rozszerzenia implementacji o nowe klasy dostarczające inną implementacje korzystania z bibliotek.

W projekcie została użyta biblioteka libUSB, głównym motywatorem użycia tej biblioteki był fakt, iż pozwala na implementacje kodu z jej użyciem zarówno pod systemami Mircrosoft Windows jak i tymi z rodziny Unix. Dodatkowym atutem była prostota dostarczonego przez nią API. Biblioteka libUSB posiada interfejs napisany bardzo intuicyjnie co znacznie ułatwiło prace podczas projektu.

Po dogłębnym zapoznaniu się z teorią oraz po zebraniu wszystkich wymagań przystąpiono do tworzenia oprogramowania z wykorzystaniem wyżej wymienionej biblioteki. Początkowym założeniem było osiągniecie prędkości przesyłu danych nie mniejszej niż 140 Mbit/s co daje 17,5 MB/s. Założenie to jest jak najbardziej możliwe z wykorzystaniem możliwości standardu USB2.0 jak i nowszych jego odsłon. Dokładniejszy opis tych zależności został opisany w rozdziale VII. Projekt został zaimplementowany w dwóch wersjach: synchronicznej oraz asynchronicznej. Wersja synchroniczna korzysta z blokujących funkcji biblioteki libUSB, natomiast asynchroniczna korzysta z nieblokujących funkcji biblioteki. W wypadku klasy odpowiedzialnej za asynchroniczne przesyłanie danych to dodatkowo całość obsługi wątków została dodana do implementacji klasy. Celem napisania tej klasy, oprócz wersji synchronicznej było sprawdzenie możliwości uzyskania lepszych wyników, a dokładniej zestawienie jednych i drugich wyników oraz porównanie ich wzajemnie celem potwierdzenia ich jakości. Istnieje możliwość łatwego rozwoju zaimplementowanej aplikacji o nowe

sposoby wysyłania danych. Wystarczy tworząc nowy sposób przesyłania danych przeciążyć podstawową klasę, zawierającą podstawową inicjalizację libUSB, zaimplementować metodę odpowiadającą za przebieg testu. Dodatkowo w wypadku bardziej złożonej inicjalizacji należy również zadbać o przeciążenie metod inicjalizacyjnych (oraz metod zwalniających zasoby) w odpowiedni sposób.

Projekt jest udostępniony jako open source i może być w dalszym ciągu rozwijany przez ludzi chcących testować przesył danych po USB implementując różnego rodzaju wariacje sposobów przesyłu danych lub mają pomysł na rozbudowanie aplikacji. Całość została udostępniona włączając mechanizmy umożliwiające odpowiednie rozróżnienie kompilacji na platformę Windowsową oraz Linuxową. Dla platformy Linux został stworzony specjalny skrypt uruchamiający odpowiednio napisane pliki makefile[1]. Zaletą projektu jest łatwość kompilacji na obie te platformy.

Wyniki przedstawione w projekcie znacznie odbiegają od wartości oczekiwanych. Spowodowane jest to brakiem obsługi USB2.0 po stronie płytki LandTiger (zaprezentowanej w rozdziale V), a dokładniej na jej chipie. Innymi słowy pomimo fizycznie wlutowanego złącza odpowiadającemu standardowi USB2.0 płytka obsługuje jedynie USB1.1 a dokładniej bufor danych wielkości jedynie 64B. Oznacza to, że w jednym momencie czasowym płytka może odebrać (lub wysłać) maksymalnie 64B danych. W rozdziale VII zostały przedstawione wyniki testów, wraz z dogłębną analizą oraz wnioskami dla każdego z trybów oddzielnie, które rozwiewają wszelkie wątpliwości odnośnie uzyskanych prędkości przesyłu danych. Założenie projektowe nie zostało wykonane do końca z powodów wyżej wymienionych, ale na podstawie uzyskanych danych można założyć iż test z wykorzystaniem płytki wspierającej standard USB2.0 (lub nowszy) będzie jak najbardziej pozytywny.

[1] makefile - plik ułatwiający budowanie oraz linkowanie plików źródłowych

Dodatek A

Zawartość płyty CD dołączonej do pracy

1. Kody źródłowe wraz ze skryptami przeznaczonymi do budowania
    - Źródło klasy Mode,
    - Źródło klasy SynchMode,
    - Źródło klasy AsynchMode,
    - Źródło klasy DebugPrinter,
    - Skrypt pozwalający na kompilacje pod systemami Unix (+UT).
    - Skrypt umożliwiający uruchomienie wielokrotnych testów
2. Niezbędne biblioteki aby uruchomić projekt pod systemem Microsoft Windows
3. Dokumentacja użytkownika oraz kodu
4. Ostateczna wersja pracy dyplomowej

Dodatek B

Wybrane wyniki przesyłu danych wygenerowane za pomocą USB benchmark

Refreshing devices.
Refreshing devices. [Done]
Starting benchmark.
Bechmarking G:(048220900002000000C3)
Started at 2015-01-31 03:03:51
...
4k:1280 Read Speed:5,26MB/s
4k:1280 Read Speed:5,09MB/s
4k:1280 Read Speed:5,17MB/s
4k:1280 Average Read Speed:5,18MB/s
2k:2560 Write Speed:2,55MB/s
2k:2560 Write Speed:2,63MB/s
2k:2560 Write Speed:2,60MB/s
2k:2560 Average Write Speed:2,59MB/s
2k:2560 Read Speed:2,73MB/s
2k:2560 Read Speed:2,75MB/s
2k:2560 Read Speed:2,68MB/s
2k:2560 Average Read Speed:2,72MB/s
1k:5120 Write Speed:1,35MB/s
1k:5120 Write Speed:1,24MB/s
1k:5120 Write Speed:1,26MB/s
1k:5120 Average Write Speed:1,28MB/s
1k:5120 Read Speed:1,31MB/s
1k:5120 Read Speed:1,41MB/s
1k:5120 Read Speed:1,26MB/s
1k:5120 Average Read Speed:1,33MB/s
Deleting file.
Benchmark done.
Ended at 2015-01-31 03:07:36

Bibliografia

[1] RS232C picture [online]
http://image.rakuten.co.jp/menet/cabinet/k_02/krs10107k_ma.jpg?_ex=60x60

[2] RS232 family picture [online]
https://upload.wikimedia.org/wikipedia/commons/thumb/2/20/VGA-RS-232C-Parallelbus.jpg/369px-VGA-RS-232C-Parallelbus.jpg

[3] Don Anderson *Universal Serial Bus System Architecture* USA: MindShare, Inc., 1997.

[4] USB Implementers Forum, Inc., USB2.0 specification reference [online]
http://www.usb.org/developers/docs/usb20_docs/

[5] USB Implementers Forum, Inc., USB3.0 and USB2.0 specyfication reference [online]
http://www.usb.org/developers/docs/

[6] Wooi Ming Tan *Developing USB PC Peripherals* Annabooks, 1999.

[7] USB logo picture [online]
http://hexus.net/media/uploaded/2013/4/e8fdbf0b-6c38-427c-931b-fa4b728856e7.png

[8] Clean Signal example [online]
http://eecatalog.com/usb/files/2012/06/120606_lecroy_1.jpg

[9] Recived Signal with noise [online]
http://eecatalog.com/usb/files/2012/06/120606_lecroy_2.jpg

[10] Received Signal after cleaning [online]
http://eecatalog.com/usb/files/2012/06/120606_lecroy_3.jpg

[11] Set of bits without encoding [online]
http://m.eet.com/media/1158450/tek8b-10bfig1.jpg

[12] Set of bits with encoding [online]
http://m.eet.com/media/1158451/tek8b-10bfig2.jpg

[13] Sending data via PHY layer [online]
http://eecatalog.com/usb/files/2012/06/120606_lecroy_4.jpg

[14] Mapping 8b to 10b [online]
http://www.latticesemi.com/~/media/LatticeSemi/Documents/ReferenceDesigns/1D/8b10bEncoderDecoder-Documentation.pdf?document_id=5653

[15] Wikimedia, 5b/6b and 3b/4b table [online]
https://en.wikipedia.org/wiki/8b/10b_encoding#5b.2F6b

[16] Frame in 128b/132b encoding [online]
http://www.synopsys.com/Company/Publications/DWTB/PublishingImages/dwtb-q4/dwtb-q414-usb3-fig1.jpg

[17] libusb 2012-2015, libusb description reference [online]
http://libusb.info/

[18] Microsoft 2015, winusb description reference [online]
https://msdn.microsoft.com/en-us/library/windows/hardware/ff540196(v=vs.85).aspx

[19] Microsoft 2015, *Developing Windows applications for USB devices* [online]
https://msdn.microsoft.com/en-us/library/windows/hardware/dn303342(v=vs.85).aspx

[20] Microsoft 2015, *How to Access a USB Device by Using WinUSB Functions* [online]
https://msdn.microsoft.com/en-us/library/windows/hardware/ff540174(v=vs.85).aspx

[21] Microsoft 2010, *How to Use WinUSB to Communicate with a USB Device* [doc from Microsoft web]
http://download.microsoft.com/download/9/C5/9C5B2167-8017-4BAE-9FDED599BAC8184A/WinUsb_HowTo.docx

[22] Pete Batard/Akeo 2013-2015 *Zadig* [online]
http://zadig.akeo.ie

[23] libusb 2012-2015, libusb documentation reference [online]
http://libusb.sourceforge.net/api-1.0/

[24] mbed, LandTiger description reference [online]
https://developer.mbed.org/users/wim/notebook/landtiger-baseboard/

[25] LandTiger with description [online]
https://dl.dropboxusercontent.com/u/5651857/landTiger3.jpg

[26] Michael J. Pont *Embedded C* , UK: Pearson Education Limited, 2002.

[27] Steven F. Barret and Daniel J. Pack *Embedded Systems* USA: Pearson Education, Inc., 2005.

[28] Rajaram Regupathy *Bootstrap Yourself with Linux-USB Stack: Design, Develop, Debug, and Validate Embedded USB* Course Technology, 2012.

[29] USB Flash Benchmark [online]
`https://www.raymond.cc/blog/download/did/1923/`